tanulmányok 125/1981

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

SOFTWARE IMPLEMENTÁCIÓS NYELVEK

Irta:

Fabók Julianna

Tanulmányok 125/1981

A kiadásért felelős:

DR. VÁMOS TIBOR

ISBN 963 311 122 6

ISSN 0324-2951

Készült a KSH SZÁMOK -ban
194/81

TARTALOMJEGYZÉK

Oldal

Oldal

Oldal

1. BEVEZETÉS

Jelen dolgozat a programozási nyelvek egy szükebb csoportjáról, a software implementációs nyelvekről kiván áttekintést nyujtani. Először is rögzitjük, hogy mit értünk software implementációs nyelv alatt. A software implementációs nyelv olyan programozási nyelv, amely software rendszerek implementálására szolgál.

Definiciónk minden szavának jelentősége van.

1/ A "rendszer" szó jelzi, hogy nagy, összetett programcsomagról van szó.

2/ A "software" rendszer olyan programcsomag, amellyel a csupasz hardware-t felruházzák, hogy a felhasználó minél kényelmesebben dolgozhasson vele. Tehát a software rendszernek feltétlenül vannak egészen gépközeli részei is.

3/ Az "implementálás" szó azt jelzi, hogy a rendszerkészitès két nagy fázisa, a tervezés és az implementálás (a kódolás) közül csak az utóbbit, a kódolást lehet az adott nyelven végezni. A tervek leirására másfajta nyelvek, az ugynevezett specifikációs nyelvek szolgálnak.

Ez a definició majdnem megegyezik a rendszerprogramozási nyelvek szokásos értelmezésével, azonban egy kicsit eltér tőle, mert a klasszikus rendszerprogramozási nyelvek a programkészitès mindkét fázisát megpróbálták lefedni vagy legalábbis nem mondták ki nyiltan, hogy csak az implementálásra szolgálnak.

Hasonlóképpen ez a definició majdnem teljesen megegyezik a MOL nyelvek (Machine Oriented Language) és a real-time nyelvek szokásos definiciójával, de azoknál a hangsuly elsősorban a gépközelségen van. A továbbiakban a software implementációs nyelveket röviden SIL nyelveknek nevezzük.

4/ Definiciónkban nincs benne, hogy "magas szintü" nyelvek, tehát mi nemcsak vagy nem elsősorban a rendszerprogramozásra alkalmas magas szintü nyelvekkel foglalkozunk, hanem bármilyen nyelvvel, amely alkalmas software készitésre, természetesen az assembly nyelveket kivéve.

Intézetünkben a közelmultban számos software rendszer készült és remélhetőleg fog is készülni. Igy a software implementációs nyelv kérdése számunkra élő kérdés. Intézetünkben ki is alakult több software implementációs nyelv (pl. PLM, GESAL).

A Programozási Rendszerek Osztály 1978 tavaszán szeminárium sorozatot tartott , ahol több software implementációs nyelvet áttekintettünk, különös tekintettel a hazánkban és az Intézetünkben honos nyelvekre. Az áttekintett nyelvek a következők voltak: PL360, XXPL, C-nyelv, GESAL, IMP, BCPL, BLISS, PROCOL, Concurrent PASCAL, EUCLID, MARY, CDL és ALEPH.

A szeminárium sorozat tapasztalatait szeretnénk most közkinccsé tenni ebben a tanulmányban. A felsorolt nyelvek közül néhányat, mint nem software implementációs nyelvet, elhagytunk, viszont néhányat hozzávettünk, ami különböző okok miatt nem szerepelt az eredeti programban. Itt elsősorban a PLM/R10-et, a MODULA-t és a TARTAN-t kell megemliteni.

Dolgozatunkban először áttekintjük a software implementációs nyelvekkel szemben támasztott követelményeket. Ezután a SIL nyelvek nyelvi jellemzőit tárgyaljuk, elsősorban

- a programszerkezet,
- az adatszerkezet és
- a vezérlési szerkezetek

szempontjából.

Végül a tárgyalt SIL nyelvek rövid áttekintése következik. Igy ezekről a nyelvekről mintegy horizontális és vertikális áttekintést is adunk. A tárgyalt nyelvek a következők:

alacsony szintü gépközeli nyelvek:

PL360, PLM/R10

középszintü adatstruktura nélküli nyelvek:

BLISS, BCPL

középszintü tipusos nyelvek:

XXPL, IMP, C-nyelv, GESAL, PASCAL

magas szintü nyelvek:

MARY, MODULA, TARTAN

Az olvasóról feltételezzük, hogy járatos az általános rendeltetésü programozási nyelvek területén, azaz ismer egy-két általános rendeltetésü programozási nyelvet (pl. ALGOL-60, FORTRAN, PL/1) és az ott szereplő fogalmakat jól ismeri. Néhány alapfogalmat azonban mégis definiálunk, elsősorban a terminológiai zavar elkerülése végett.

A tárgyalás során a hatékony implementáció szempontjait nagy sullyal figyelembe vesszük.

2. A SIL NYELVEKKEL SZEMBEN TÁMASZTOTT KÖVETELMÉNYEK

Az implementációs nyelv tulajdonképpen feladatorientált programozási nyelv, amely a software rendszerek készitésekor (kódolásakor) felmerülő fogalmak, módszerek, adatstrukturák leirását teszi lehetővé. Ennek megfelelően az implementációs nyelvekkel szembeni követelményeket két fő csoportba sorolhatjuk. Egyrészt elégitsék ki az általános programozási nyelvekkel szemben támasztott követelményeket, másrészt tegyék lehetővé a software készitésnél használt speciális programozási technikák alkalmazását és speciális igények kielégitését.

A programozási nyelvekkel szemben általában támasztott követelmények:

a. Problématérhez való közelség
   A nyelv tartalmazza azokat a fogalmakat, amelyekre az adott felhasználási területen szükség van.
b. Biztonság, definiáltság, zártság
   Az elkövetettt hibák compile-time felderitése, az undefined esetek definiálása, hibához vezető lehetőségek kizárása (pl. tömbindex tulcsordulás stb.).
c. Olvashatóság, javithatóság, öndokumentálás
   A program mások által könnyen olvasható és javitható legyen, mintegy saját maga dokumentációjaként szolgáljon.
d. Egyszerüség, tanulhatóság
   A programozó is ember, tul bonyolult dolgokat nehezen tanul meg.
e. Modularitás, részenkénti fordithatóság.
   Nagy programot sok ember készit, lehetőség kell a részek összerakására.
f. Portabilitás, gépfüggetlenség.
g. Rugalmas I/O kezelési lehetőség.

Az implementációs nyelvekkel szemben támasztott speciális követelmények:

a. Gépközelség
   A hardware által nyujtott összes lehetőség kihasználása.

b. Hatékonyság

Kis memória kapacitás, gyors futási idő a készitett programoknál.

Sokan a programozási nyelvekkel szemben támasztott követelmények közé sorolják a kiterjeszthetőséget és a strukturáltságot is. Véleményünk szerint ezek csak eszközök a fenti célok elérésére. Igy a kiterjeszthetőség eszköz a problématérhez való közelség és az egyszerüség, tanulhatóság közötti ellentmondás áthidalására. A strukturáltság pedig az olvashatóságot, öndokumentálást segiti elő.

3. A SIL NYELVEK NYELVI JELLEMZŐI

Az áttanulmányozott nyelvek meglepően sok vonásban hasonlitanak egymásra. Néhány sajátságban viszont eltérnek egymástól. Legfontosabb nyelvi jellemzőiket a programszerkezet, az adatszerkezet és a vezérlési szerkezetek milyensége mutatja. Ebben a fejezetben ezeket a jellemzőket tárgyaljuk általánosan,összefoglalva a nyelvek közös vonásait. Kétféle szempontból értékeljük ezeket, egyrészt a programozó kényelme, másrészt a hatékony implementáció szempontjából.

3.1 Programszerkezet

Programszerkezet szempontjából a rendszerprogramozási nyelvek nem különböznek lényegesn az általános rendeltetésü nyelvektől, kivéve azokat a tulajdonságokat, amelyeket a nagy méret és a hatékony implementálás megkövetel. Programszerkezet alatt azt értjük, hogy mi a program elemi egysége és hogyan kell a nagyobb egységeket a kisebb egységekből vagy az elemi egységekből felépiteni.

Tulajdonképpen a program legkisebb egységei a kulcsszavak, azonositók, konstansok, müveletjelek és elhatárolójelek. Az utasitásokat a nyelv szintaxisának megfelelően kell a felsorolt részekből összerakni. Azonban az utasitások szerkezete nem tartozik a programszerkezetbe. Hasonlóan mint ahogy a kémiai szerkezetbe is csak a molekulának atomokból való felépitése tartozik, magának az atomnak a belső szerkezete már nem.

Az elemi egység az utasitás, egyes nyelvekben a kifejezés. Közbülső programegység az összetett utasitás és a blokk, a függvény és a szubrutin, valamint a modul. Még nagyobb programegység a szegmens. Egy vagy több szegmens együttesen alkotja a programot. A modul és a szegmens szavak szóhasználatával kapcsolatban megjegyezzük, hogy a modul szót a MODULA nyelv modul fogalmának megfelelően használjuk (lásd 3.1.5).

A közbülső programegységek egyik legfontosabb funckiója a nevek hatáskörének, scope-jának szabályozása. Egy programegy-

ségben deklarált és másutt felhasznált nevet exportált névnek, másutt definiált és itt felhasznált nevet pedig importált névnek nevezzük. A programegységek scope szempontjából viselkedhetnek nyitott vagy zárt scope-ként.

A nyitott scope automatikusan importál minden nevet az őt körülvevő scope-ból, de semmit sem exportál. Ilyen az ALGOL-60-ból jól ismert blokkstruktura a globális, lokális változókkal. Szemléletesen a nyitott scope-ot félig áteresztő hártyához hasonlithatjuk.

Zárt scope esetén az export/import tevékenység programból szabályozható. Importálhatók olyan azonositók, amelyeket más egységek exportáltak és exportálhatók az egységben definiált nevek. A nem exportált azonositók a környezet számára rejtettek maradnak. A modul zárt scope-ot alkot.

Az eljárás és függvénytörzs némely nyelvben nyitott scope, pl. ALGOL-60, más nyelvekben zárt scope pl. FORTRAN.

A továbbiakban a programegységeket és azok összekapcsolásának módját tárgyaljuk.

3.1.1 Az elemi egység

Az elemi egység az utasitás, amely lehet direktiva, deklarativ utasitás vagy végrehajtható utasitás. A direktiva magának a fordítónak szóló utasitás. Pl. feltételes fordítás. A deklarativ utasitások tartalmazzák az adatok leirását, lehetnek definiciók vagy deklarációk. A végrehajtható utasitások tartalmazzák az akciók leirását és lehetnek értékadás, vezérlés átadás vagy rutin hivás. Az értékadó utasitás jobb oldalán álló kifejezés szintén akciókat jelöl ki, kijelöli, hogy milyen müveleteket és milyen sorrendben kell az operandusokon elvégezni. Az akciók elvégzése, azaz a kifejezés kiértékelése következtében érték jön létre. Az utasitásnak hatása, a kifejezésnek értéke van.

Néhány modern nyelvben az értékadás, az átutalás is operátor, igy az értékadó utasitás egyszerüen kifejezéssé válik. Az A→B kifejezésnek értéke is és hatása is van. Értéke az A érték, hatása pedig az, hogy a B változó felveszi az A értéket. Az i-

lyen nyelvekben az akciókat kifejezések sorozata irja le.

Ez a nyelvi sajátság lehetővé teszi azon hardware lehetőség kihasználását, hogy tárolás után az érték az akkumulátorban marad és tovább lehet veleszámolni. Pl. A+B → C+D kifejezés értéke A+B+D, hatása pedig az, hogy a C változó felveszi az A+B értéket. A program kifejezések sorozata, amelyek kiértékelése szekvenciálisan történik, hacsak a kifejezések hatásai nem gondoskodtak az értékek tárolásáról. Ilyen nyelvek pl. a BLISS, C-nyelv, GESAL, MARY. A BLISS-ben és a MARY-ben a vezérlési szerkezetek is kifejezések olyan értelemben, hogy értékük és hatásuk van. Hatásuk az előirt vezérlésátadás, értékük pedig

- a vezérlési szerkezetből értelemszerüen adódó érték. Pl. az IF szerkezet két kifejezés közül, a CASE szerkezet pedig több kifejezés közül választja ki, hogy melyik kerül kiértékelésre, ennek értéke lesz a vezérlési szerkezet értéke is. A függvényhivás értéke értelemszerüen a visszatérő érték.
- mesterségesen kijelölt érték. Pl. a BLISS-ben a ciklus szerkezetek értéke = -1;
- a pillanatnyilag aktuális érték. Pl. a MARY-ben, ha kifejezés sorozat kiértékelését szakitja meg egy GOTO utasitás.

A MARY-ben kifejezésen belül is szerepelhet cimke.

Példa:

```
A+B      L:  +C → D
              :
              :
2*C           GO  TO L
```

Amikor a program végrehajtása elölről történik, akkor D=A+B+C lesz. A GOTO utasitás kiértékelése után pedig D=3*C.

A kifejezésorientált nyelv nagyobb szabadságot ad a programozónak a nyelv bonyolultságának lényeges növelése nélkül. Emellett a tárgykód hatékonyságát is növelni lehet.
Tehát SIL nyelvekként kifejezésorientált nyelvek használata ajánlott.

A továbbiakban az elemi egységet mindig utasitásnak hivjuk, akár utasitás, akár kifejezés.

3.1.2 Az összetett utasitás

Az összetett utasitás utasitásoknak utasitás zárójelbe tett sorozata. Az utasitás zárójel általában vagy egyszerü zárójel vagy a BEGIN...END kulcsszavak, de előfordulhat más kulcsszó is, pl. START...FINISH. Az összetett utasitás fogalmára akkor van szükség, ha a vezérlési szerkezetek olyanok, hogy bizonyos pozicióban csak egy utasitás állhat. Ilyen vezérlési szerkezetek például

```
IF     F1  THEN  U1   ELSE  U2 ;
FOR    N1  FROM  K1   TO    K2  BY  K3  DO  U1 ;
WHILE  F1   DO   U1;
```

Itt F1 feltétel, U1 és U2 utasitás, K1, K2 és K3 aritmetikai kifejezés és N1 változónév. Ezekben a vezérlési szerkezetekben U1 és U2 helyén csak egy utasitás állhat. Ha ide több utasitást kivánunk irni, akkor utasitás zárójelet kell használnunk. Összetett utasitás van pl. a PL360, PASCAL, BCPL, BLISS, C-nyelv nyelvekben.

Ha a vezérlési szerkezetek strukturája nem követeli meg, hogy bizonyos pozíciókban csak egy utasitás álljon, akkor a nyelvben nincs szükség összetett utasitásra. Mint 3.3-ban ezt látni fogjuk, ezt a problémát leggyakrabban ugy oldják meg, hogy a vezérlési szerkezet záró kulcsszóval rendelkezik. Pl. IF...FI, DO...OD, CASE...ESAC. Ilyen nyelvek pl. a GESAL, XXPL. Az összetett utasitás fogalmának elhagyása világosabbá teszi a nyelv szerkezetét és a forditóprogram dolgát is megkönnyiti.

A változók scope-ja szempontjából az összetett utasitás kétféleképpen viselkedhet.

a/ A változók scope-jára nincs befolyással. Ez esetben a következő problémák merülnek fel:
   - Ha az összetett utasitás nem külön scope, akkor az összetett utasitásba kivülről be lehet ugrani. Ha speciális pozíciókban pl. FOR ciklusnál ezt nem akarjuk megengedni, akkor külön meg kell tiltani a ciklus törzsbe való beugrást. Pl. ALGOL-60.

- Neheziti az implementálást, ha az összetett utasitás is és a blokk is a BEGIN ... END kulcsszavak között áll, bár scope szempontjából eltérő a viselkedésük. Ekkor BEGIN után vizsgálni kell a következő utasitást. Ha deklaráció következik, akkor blokkról van szó és a cimkék scope-ja is megváltozik. Ha nincs deklaráció, akkor összetett utasitásban vagyunk és a scope nem változik.

b/ Az összetett utasitás nyitott scope. Ez esetben BEGIN után mindig scope változás van és automatikusan adódik, hogy összetett utasitás belsejébe, igy FOR, WHILE stb. ciklusok törzsébe nem szabad ugrani. Érdekes, hogy ezt a kézenfekvő lehetőséget sem a hagyományos programozási nyelvek, sem az összetett utasitással rendelkező SIL nyelvek nem használják ki. A.N.Habermann többek között ezt is a PASCAL szemére veti [17].

3.1.3 Blokk

A blokk utasitászárójelbe tett olyan programrész, amely deklarativ utasitásokat is tartalmaz. Deklarációt mindig kell tartalmaznia, különben nem blokk, hanem összetett utasitás. Blokkstrukturával a változók hatáskörét korlátozhatjuk. Egy blokkban deklarált változó hatásköre mindig lokális a blokkra nézve. Egymásba skatulyázott blokkok esetén a külső blokk változói globálisak a belső blokkra nézve, azaz automatikusam importálódnak, de a belső blokk változói lokálisok, azaz nem exportálódnak. Más szavakkal a blokk nyitott scope-ot alkot. Ujradeklarálás lehetséges,ez esetben az ujradeklarált változó lokális saját blokkjában, kivül pedig az előző deklaráció érvényes.

Blokkstruktura esetén a tárgyprogramban overlay-ezett tárkezelés valósul meg, ugyanis a párhuzamos blokkoknak ugyanarra a memória területre foglalhatunk helyet. Igy a program memória igénye lecsökken. A helyfoglalás azonban statikus, az adatok helye már a forditás során meghatározódhat (feltéve, hogy nincs

dinamikus tömbindex és rekurziv rutinhivás). A változók értékének elérése viszont bonyolultabbá válik. Hiszen, ha egy változó a végrehajtás alatt lévő blokkban nincs lekötve, akkor az őt tartalmazó legkisebb külső blokkban kell keresni. Ha ott sincs lekötve, akkor a még külsőbb blokkban és igy tovább a legkülsőbb szintig. Azonban mindez a forditás ideje alatt történhet, s igy a tárgykód hatékonyságát nem rontja.

Megjegyezzük, hogy itt szándékosan nem beszélünk dinamikus tárkezelésről. A dinamikus tárkezelés azt jelenti, hogy a változók helye futáskor foglalódik és szabadul fel. A dinamikus tárkezelést megvalósitó verem technika lehetővé teszi

- a blokkstruktura,
- a rekurziv rutinhivás és
- a dinamikus tömbök

egyidejü megvalósitását. Azonban ha csak a blokkstrukturát nézzük önmagában, akkor a fentebb mondottak érvényesek.

Programozási szempontból a blokkstruktura megkönnyitheti a különböző személyek által irt programok összeépitését, hiszen az azonos elnevezésü változók nem okozhatnak bajt az összeépitésnél. Másrészt viszont hátrányt jelent, hogy az egyes blokkok közötti adatátadás csak globális változókon keresztül történhet és ezt a programozónak kell megszervezni. Az is hátrányt jelent, hogy az egyes blokkok összeszerkesztését forditás előtt forrásnyelven kell elvégezni. Ezért a különböző személyek által irt programok összeépitésére kényelmesebb a szegmens szerkezetet használni, amely megengedi a részenkénti fordithatóságot is (3.1.6).

A blokkstruktura nem segiti elő a top-down, szintről-szintre fokozatosan bővitő program tervezést. A program öndokumentáltsága, olvashatósága is rossz, mert a blokkok teljes terjedelmükben a helyükön vannak és nincs, illetve nem tartozik a nyelvhez a program vázát mutató rövid leirás, amelynek kifejtésével áll elő a teljes program.

Megjegyezzük, hogy a blokkstruktura értékelésére vonatkozó nézetünk ellentétes több neves számitástechnikai szakember véleményével. Pl. N.Wirth irja: "Block structure has proved to be a most valuable facility in systematic program design." (A blokk-

struktura bizonyult a szisztematikus program tervezés egyik legértékesebb eszközének) [23] 5. old. Véleményünk szerint nem a blokkstruktura, hanem a változók scope-jának a korlátozása a lényeges, ami más eszközökkel is elérhető. Pl. a PASCAL-ban az eljárástörzs játssza ugyanezt a szerepet.

3.1.4 Eljárás (szubrutin) és függvény

Mindkettő önálló programrész, amely a hivó programmal a paramétereken keresztül érintkezik. A különbség az közöttük, hogy a függvénynek egyetlen visszatérő paramétere van, továbbá, hogy a rutinhivás akció, tehát önálló utasitás, mig a függvényhivás egy értéket képvisel, tehát kifejezés operandusa lehet. További különbség lehetne, hogy a függvénynél nem kivánatos a side-effect, ezért a globális változók használata kerülendő. Emiatt a függvénytörzsnek zárt scope-nak kellene lenni, mig az eljárás nyitott scope lehet. A gyakorlatban azonban ezt a megkülönböztetést általában nem teszik meg és scope szempontjából egyformán kezelik őket. A továbbiakban a két fogalmat közösen tárgyaljuk és eljárás néven beszélünk róla.

A paraméterátadás formái a hagyományos programozási nyelveknél az érték, a hivatkozás és a név szerinti paraméterátadás. Pl. az ALGOL-60-ban név és érték szerinti paraméterátadás van, a FORTRAN-ban hivatkozás szerinti. Mint ismeretes, érték szerinti paraméterátadás esetén az aktuális paraméterek értéke bemásolódik az eljárás lokális adatmezejébe. Ezért összetett adatmezők, pl. tömbök érték szerinti átadását általában célszerü megtiltani. Ezeket csak hivatkozás szerint szabad átadni. Ha a nyelvben pointer változó van, akkor egy adatmezőre mutató pointer-változó értékének érték szerinti átadása azonos hatásu az adatmező hivatkozás szerinti átadásával. Ezért az érték szerinti átadás tulajdonképpen minden célt kielégit, s igy hivatkozás szerinti paraméterátadás nem is szükséges. Ennek ellenére néhány rendszerprogramozási nyelv mégis megtartotta. Pl. IMP, PASCAL stb. Név szerinti paraméterátadás a SIL nyelvekben nincs.

Az eljárások vagy egymásbaágyazhatók vagy nem, és vagy rekurzivak vagy nem. Egymásbaágyazáson azt értjük, hogy egy eljá-

rás tartalmazza egy másik eljárás deklarációját. Rekurziv eljárásról pedig akkor beszélünk, ha egy eljárás az eljárástörzsön belül önmagát hivja közvetve vagy közvetlenül.

	egymásba ágyazhatók	nincs egymásba ágyazás
rekurziv	ALGOL-60	GESAL
nem rekurziv	XXPL	FORTRAN

A rekurzió nagyon megnöveli egy nyelv erejét. Rekurziv eljárásokat majdnem minden SIL nyelv megenged. Az eljárások scopejával kapcsolatban eltérő a gyakorlat. Egyes nyelvekben az eljárástörzs nyitott scope, azaz a törzsön belül megengedik a globális változók használatát. Pl. GESAL, PASCAL, más nyelvek viszont nem. Pl. TARTAN.

Érdekes jelenség, hogy a MODULA-ban az eljárás attól függően nyitott vagy zárt scope, hogy van-e benne "use-list". Ha van, akkor zárt scope, különben nyitott.

Ha az eljárástörzs nyitott scope, akkor egymásbaágyazott eljárások esetén a nevek hatásköre ugyanolyan, mint blokkstruktura esetén. Tehát egymásbaágyazott eljárásokkal mindent meg lehet csinálni, amit blokkstrukturával lehet, azonban a program sokkal olvashatóbb lesz, mert a főprogram mutatja a program vázát, azaz az egymásután hivandó eljárásokat.

3.1.5 Modul

A modul szót különböző nyelvek különböző értelmben használják. Szokásos használata, hogy a modularitás alatt a részenkénti fordithatóságot értjük. Ez esetben a modul a programnak egy önállóan forditott része. Mi erre a fogalomra a szegmens kifejezést használjuk. Szóhasználatunkban a modul zárt scope-ot alkotó programegység. Ez a terminológia a MODULA és TARTAN modul fogalmának felel meg.

A modul fogalom legfontosabb jellemzői. A modul adatdeklarációkból, eljárásdeklarációkból és végrehajtható utasitások so-

rozatából álló egység, amely zárt scope-ot alkot. Legfontosabb funkciója, hogy a nevek hatáskörét szabályozza. N.Wirth irja "The module should be thought of as a fence around its objects." (A modult ugy kell elképzelni, mint egy keritést az objektumai körül.) [23] 5. old. vagy másutt: "The module has one and only one function, namely to establish a static scope of identifiers" [25] 69. old. (A modulnak egy és csakis egy szerepe van, tudniillik az azonositók egy statikus scope-jának a létrehozása.) A modulon belül a nevek lokálisak, kivéve az exportált neveket, amelyeket a define-list tartalmaz. Globális változó nem használható, kivéve az importált változókat, amelyeket az use-list tartalmaz. Tehát a define-list, use-list használatával a programozó szabályozni tudja a változók hatáskörét. Exportálni lehet bármilyen nevet, változó, konstans, tipus és eljárás nevet is.

A modul fogalom két őse az ALGOL-60 own változója és a SIMULA-67 osztály fogalma. A modul számára akkor foglalódik hely, amikor deklarációja feldolgozásra kerül, azaz amikor az őt tartalmazó eljárást meghivták és a modul törzse is ekkor fut le. A modul objektumai azonban a törzs lefutása után is életben maradnak és az exportált nevek hivatkozhatók. A SIMULA-67 osztály fogalmától eltér annyiban, hogy a modul mindig egypéldányu és deklaráláskor generálódik. Hasonlit annyiban, hogy a modulban adatstrukturát és rajta értelmezett eljárásokat lehet leirni és kivülről a definiált eljárások segitségével kezelhetjük az adatstrukturát annak konkrét ismerete nélkül. Ezzel a technikával megvalósitható az információk "elrejtése", absztrakt adatstruktura kezelése.

Példa: Ha egy modulban egy queue-t akarunk kezelni, akkor definiáljuk a queue adatstrukturáját (egy tárterület két mutatóval) és az inqueue, dequeue eljárásokat, de csak a tipusnevet és az eljárásneveket exportáljuk. A tipusnév exportja nem jelenti a tipus leirás exportját is. Felhasználáskor az importált tipushoz deklarálunk változókat és ezekre alkalmazzuk az importált eljárásokat. A modulok akárhány mélységben egymásbaágyazhatók.

3.1.6 Szegmens

A szegmens szerkezet arra szolgál, hogy különböző programozók esetleg más-más nyelven irt programjait egyetlen programmá lehessen összeállitani. A szegmens egy teljesen önálló program egység. A szegmensek közötti kapcsolat közösen használt objektumokon keresztül valósul meg. Ezeket az objektumokat a programban általában az "EXTERNAL" - "ENTRY" kulcsszavakkal jelölik meg. EXTERNAL az a változó, amelynek helye valamely másik szegmensben van, az ENTRY változóra pedig másik szegmensből is hivatkozhatnak, de a helye itt van. Néhány SIL nyelvben az ENTRY kulcsszót nem teszik ki, hanem a főprogram változói automatikusan ENTRY-nek számitódnak. Pl. XXPL, C-nyelv. Az ENTRY és a GLOBAL kulcsszók szinonimák.

A modul és a szegmens fogalmak között sok a hasonlóság. Közös vonás az, hogy mindkettő zárt scope-ot alkot. Eltérő viszont a cél a két esetben. Modul esetén a cél az információ elrejtése, a változók scope-jának korlátozása. Ez nincs ellentétben a modulok egymásbaágyazásával. Szegmens esetén a cél a részenkénti fordithatóság. Ebből következik, hogy a szegmenseknek zárt scope-ot kell alkotni, hiszen a szegmens önállóan fordítható a többi szegmens belső világának ismerete nélkül. A szegmensek közötti kapcsolat megteremtése a linkage editor dolga. Ehhez adnak segitséget az EXTERNAL-ENTRY deklarációk. A szegmensek mindig mellérendelt szerepet játszanak, nincs egymásbaágyazás.

3.1.7 Tárkezelés

Ebben a pontban csak a programszerkezettel kapcsolatos tárkezelési problémákkal foglalkozunk, változó tömbindex miatt szükséges dinamikus tárkezeléssel nem. Blokkstruktura és egymásbaágyazható vagy rekurziv eljárások egyidejü fennállása esetén a tárkezelési mechanizmus bonyolulttá válik. A bonyolultságot az okozza, hogy a dinamikus (hivási) és statikus (hivatkozási) környezet elválhat egymástól. Hivási környezet alatt azt a környezetet értjük, amely az adott eljárást meghivta, ezzel az aktuális paramétereken keresztül érintkezik az eljárás. A hivatkozási környezet pedig az a környezet, melyben az eljárás törzse

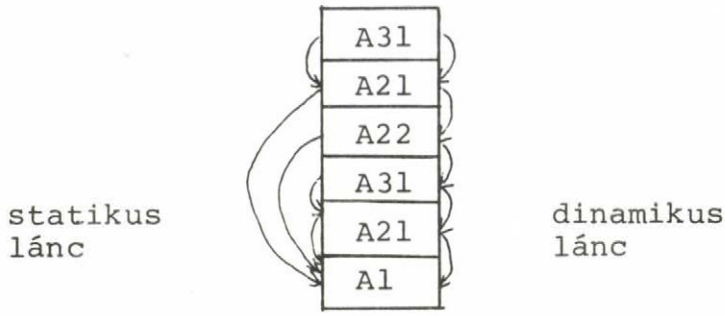
deklarálva lett és melynek változóira, mint globális változókra az eljárás törzse hivatkozhat. A kettő nem szükségképpen diszjunkt. Tekintsük a következő példát.

```
declare proc A1
     declare var B1
     declare proc A21
          declare var B21
          declare proc A31
               declare var B3
               BEGIN
               access B1VB21VB3
               call A22
               (call A31VA21VA1)
               END
          BEGIN
          access B1VB21
          call A31
          call A22
          (call A21VA1)
          END
     declare proc A22
          declare var B22
          BEGIN
          access B1VB22
          call A21
          (call A22VA1)
          END
     BEGIN
     access B1
     call A21VA22
     (call A1)
     END
```

1. ábra

A példa a nevek hatáskörét is mutatja. A zárójelbe tett hivások csak akkor végezhetők el, ha rekurzió megengedett.

Tekintsük most a következő hivás sorozatot A1 : A21 : A31 : : A22 : A21 : A31. Ez esetben a verem szerkezete



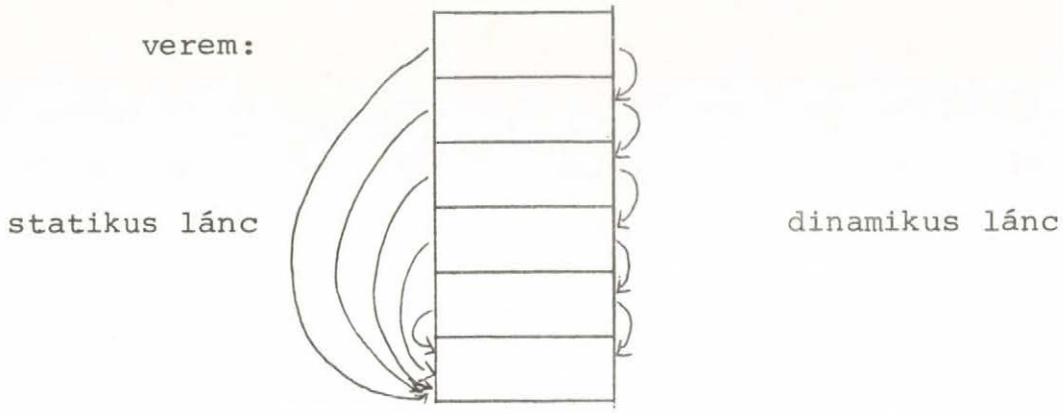
2. ábra

A verem elemei az egyes eljárások számára foglalt tárterületek. Láthatjuk, hogy pl. A21 hivója A22, statikus környezete pedig A1.

A gyakorlati megvalósitásra két módszer van. Az egyik módszer az, hogy az adatmezőket kiegészitjük két mutatóval, az egyik mutató a hivó eljárás adatmezejére mutat, a másik mutató pedig a statikus szerkezetnek megfelelő adatmezőre, azaz arra az adatmezőre, ahol a globális változók le vannak kötve. A hivó eljárásra mutató pointerek alkotják a dinamikus (hivási) láncot, mig a statikus környezetre mutató pointerek a statikus (hivatkozási) láncot alkotják. A 2. ábrán ezek is fel vannak tüntetve. A másik módszer az, hogy bevezetünk két vektort, az egyik dinamikus sorrendben tárolja a báziscimeket, a másik pedig a "mélységvektor", amely a program statikus báziscimeit mutatja.

Mindkét mechanizmus megvalósitása lényegesen lelassitja a tárgy program müködését, hiszen mindenegyes változó értékének eléréshez több lépésből álló eljárást kell végrehajtani. Fel kell göngyöliteni a statikus láncot vagy elemezni kell a mélységvektort. Ráadásul mindezt futásidő alatt kell elvégezni, hiszen a dinamikus környezet csak futásközben jön létre.

Vizsgáljuk most meg, hogyan kerülhetjük el ezt a nehézséget. A 3. ábrán látható egy egyszerübb tárkezelési mechanizmus.



3. ábra

Minden nem lokális változó ugyanott van lekötve. Létjogosultsága abban az esetben van, ha a nyelvben minden eljárás a főszinten van deklarálva, azaz megtiltjuk az eljárások egymásbaágyazását (GESAL, C-nyelv), vagy ha az eljárások csak speciális globális változókat használhatnak. (COMMON, STATIC). Pl. FORTRAN, BCPL.

Megvalósitási mód: a változók helye fordításkor meghatározható, a nem lokális változóké a statikus mezőben, a lokálisoké (a formális paraméterek is lokális változóknak számitanak) pedig az eljárás saját adatmezejében. A futtatás 2 bázisregiszter felhasználásával gazdaságos. Az egyik bázisregiszter a statikus adatmező kezdőcimét tartalmazza, a másik pedig az aktuális lokális adatmezőét. A verem kezelése a lokális bázisregiszter tartalmának megváltoztatásával történhet.

Ezt a tárkezelési mechanizmust SIL nyelveknél érdemes megvalósitani, mert hatékony implementáció készithető és nem szükiti le tulságosan a programozó lehetőségeit.

3.2 Adatszerkezet

Az adatszerkezet alatt azt értjük, hogy milyen adatok vannak a nyelvben és hogyan lehet őket használni, mit kell deklarálni, hogyan lehet elérni és használatkor milyen ellenőrzés történik.

Az adatszerkezetek helyes megválasztása a SIL nyelvek kulcskérdése. Döntő fontosságu, hogy a gépközeli dolgokat is le lehessen irni, legyen bit-manipuláció és karakter, illetve string feldolgozás.

A továbbiakban először áttekintjük a SIL nyelvek adatszerkezetével szemben fennálló követelményeket, majd foglalkozunk a jelenlegi SIL nyelvek tipikus adatszerkezeteivel. A nem tipusos nyelvek ismertetése után a tipusos nyelvekkel foglalkozunk az egyszerü önálló és származtatott, az összetett tipusok és az union tipus, valamint a tipussal kapcsolatos egyéb problémák tárgyalásán keresztül.

3.2.1 Követelmények a SIL nyelvek adatszerkezetével szemben

Véleményünk szerint a SIL nyelveknek lehetővé kell tenni,

a/ hogy a programozó uj adatszerkezetet definiálhasson, azonban beépitett tipusként tartalmaznia kell a rendszerprogramozásnál használt szokásos adatstrukturákat.

b/ hogy a programozó kézben tarthassa az adatok helyfoglalását, ugyanahhoz a memória helyhez különbözőképpen is hozzáférhessen és a változó kezelésére vonatkozóan is adhasson utasitást. (Explicit deklarációs utasitás, union tipus, memória osztály.)

c/ A gépközeli dolgok leirásához egyes adatok bitjeihez való hozzáférés szükséges.

d/ Szövegfeldolgozás végrehajtásához pedig a karakter illetve string tipusu változó bevezetése célszerü.

e/ Referencia tipusu változó kell a hivatkozások kezeléséhez, pl. táblázatok, tömbök átadásához.

f/ Növeli a biztonságot, ha szigoru tipus ellenőrzés van, bár a SIL nyelvek egy része éppen az ellenőrzést ejti el a memóriához való többféle hozzáférés érdekében.

Természetesen az itt felsorolt szempontokat nem mindenki tartja egyformán fontosnak, igy az egyik nyelvben az egyik, más nyelvben a másik szempont kap nagyobb sulyt.

3.2.2 A tipus

Az adatok tipusossága a SIL nyelvek egyik legvitatottabb kérdése volt. A tipus fogalmának leglényegesebb vonásait Hoare igy foglalja össze [28] 92. old.: "The type information determines the representation of the values of the variable, and the amount of computer storage which must be allocated to it. Type information also determines the manner in which arithmetic operators are to be interpreted; and enables a compiler to reject as meaningless those programs which invokve in appropriate operations." (A tipusinformáció meghatározza a változó értékének reprezentációját és a számára lefoglalt tárterület nagyságát. Meghatározza az aritmetikai operátorok interpretálásának módját és képessé teszi a compiler-t, hogy felfedje a programban előforduló tipusösszeférhetetlenséget.)

Hoare szerint a tipus fogalmának legfontosabb jellemzői:

1/ A tipus meghatározza az értékeknek azt a készletét, amelyet a változó vagy kifejezés felvehet.
2/ Minden értékhez tartozik egy és csak egy tipus.
3/ Bármelyik konstans, változó vagy kifejezés által jelölt érték tipusa leszármaztatható a formájából vagy a környezetéből anélkül, hogy konkrét értékét ismernénk.
4/ Minden operátor előre rögzitett tipusu operandusokat vár és rögzitett tipusu eredményt szolgáltat. Ahol ugyanazt a szimbólumot használjuk különböző tipusok esetén, pl. + jelöli az integer és real összeadást is, ott ezt a szimbólumot többértelmünek tekintjük, amely több különböző aktuális operátort jelöl. Ez a szisztematikus többértelmüség mindig a forditás során tisztázódik.

5/ Implementáláskor a tipus információt arra használjuk, hogy
- felismerjük és/vagy meggátoljuk az értelmetlen program konstrukciókat;
- meghatározzuk az adatok reprezentálásának és manipulálásának módját.

Az elmondottak az egyszerü tipusra vonatkoznak. Mint láttuk, egyszerü tipusnál a változó tipusa megszabja, hogy értékét milyen készletből veheti. Az egyszerü tipusok egy részénél, a beépitett vagy standard tipusoknál, a gépi reprezentáció egyértelmüen megszabja az értékkészletet és a végezhető müveleteket. Pl. az integer tipus értékkészlete az adott bithosszusággal ábrázolható egészszámok, a karakter tipus értékkészlete a konfigurációban használt karakterkészlet stb. Más tipusoknál csak az értékkészlet milyenségét adja meg a nyelv, pl. halmaz, hatványhalmaz stb. Ez esetben a programozónak kell megadni a konkrét készletet, amit névvel láthat el, pl. felsorolja a halmaz elemeit. Ilyenkor tipus definicióról beszélünk. A továbbiakban a felhasználó által definiált tipusokat definiált tipusoknak fogjuk nevezni.

A változó deklaráció egy konkrét tipushoz tartozó változókat sorolja fel. Helyfoglalás tipusdefiniciónál nincs, csak deklaráció esetén van. A tipus definició és a változó deklaráció nem szükségszerüen különül el élesen, össze is olvadhat. Pl. most definiált tipushoz is deklarálhatunk változót a C-nyelvben. Terminológiai zavart okoz, hogy néhány nyelv a tipus definiálást is deklarációnak mondja.

Mig az egyszerü beépitett és egyszerü definiált tipusok önálló tipusok, mert a tipus önmagában megszabja az értékkészletet, addig a származtatott tipus valamilyen alaptipusra vonatkoztatva definiál egy uj tipust. Pl. subrange, set, pointer. Az alaptipus általában már definiált tipus kell hogy legyen.

Az összetett adattipus egyszerübb adattipusokból adott szabályok szerint összeállitott adatsokaság, amelyre egységként is lehet hivatkozni, de lehet a részeire külön-külön is.

Önálló összetett adattipus a tömb és a file, ezek azonos tipusu elemekből állnak, a rekord pedig különböző tipusu adatok egységbe foglalása.

Származtatott összetett adattipus a bits, ami array of boolean [1:W] (W a szóhossz) és a string [n], ami array of character [1:n].

Eltekintve attól, hogy a korai általános rendeltetésü programozási nyelvek tipusai (integer, real, boolean) nem alkalmasak rendszerprogramozási fogalmak leirására, a tipus, a tipus függő operátorok és a tipus ellenőrzés erősen korlátozzák a programozó szabadságát. Meggátolják, hogy egy adathoz többféleképpen is hozzáférjen, ami pedig rendszerprogramozásnál nagyon kivánatos. Éppen ezért a korai rendszerprogramozási nyelvek (BLISS, BCPL) el is vetik a tipus fogalmát. Minden adat egyszerüen egy bitminta, csupán a cim és tartalma között tesznek különbséget, de azt is a programozónak kell kézben tartania. Minden adattal minden müveletet lehet végezni, az operátorok nem tipusfüggőek, a programozónak kell tudni, hogy melyik adat micsoda. Az ilyen programozás azonban rengeteg hibázási lehetőséget rejt magában. Ezért a későbbi rendszerprogramozási nyelvek felujitják a tipus fogalmát, a korai nyelvek tipusait kiegészitik a rendszerprogramozáshoz jól illő tipusokkal, pl. referencia, karakter, stringgel és a halmazszerü tipusok különféle válfajaival, enumeráció, powerset, subrange stb. Lehetőség nyilik a programozó számára uj tipus definiálására, sőt arra, hogy ugyanazt az adatot különböző szituációkban különböző tipusuként használja természetesen szigoruan ellenőrzött feltételek mellett (union tipus).

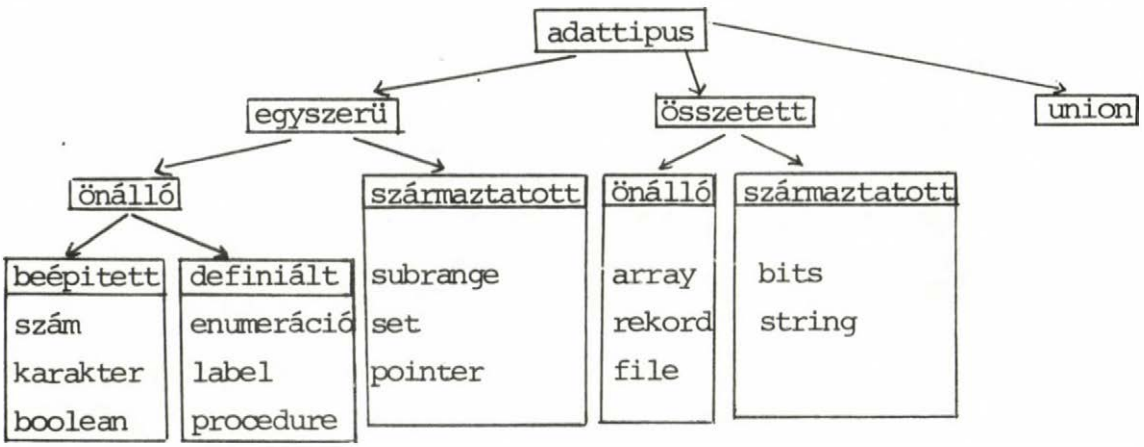
A tipusok létezése felveti a konverzió kérdését. Bizonyos szituációkban célszerü az automatikus konverzió, például integer szélesitése real-lé. Más szituációkban jó, ha a programozónak kell kérnie a transzformációt egyik tipusból a másikba pl. kerekités vagy csonkitás. Erre általában standard függvények állnak rendelkezésre. A nyelvek koerció, kényszerités szempontjából két nagy táborra oszthatók. A koercáló nyelvek élükön az ALGOL 68-al az automatikus koerciót részesitik előnyben, és pontosan kidolgozták annak a kezelését, hogy milyen szövegkörnye-

zetben (kontext) milyen tipust milyen más tipussá lehet automatikusan kényszeriteni. Ebbe a nyelvcsoportba tartozik a MARY is. A kényszerités és kontext fogalmával 3.2.10.2-ben még foglalkozunk.

A szigoru tipusellenőrzést végző nyelveknél automatikus koerció nincsen, minden koerciót a programozónak kell előirni. A SIL nyelvek között a szigoru tipusellenőrzést végző nyelvek vannak tulsulyban. Ugy tünik, hogy a fejlődés tendenciája e nyelvek javára mutat.

A továbbiakban először a nem tipusos nyelvek adatszerkezetét vizsgáljuk, majd a tipusos nyelvek sajátságaival foglalkozunk.

A tipusok csoportositására [17]-ben és [20]-ban találunk példákat. Mi egy ezektől eltérő hierarchiában tárgyaljuk a tipusokat. A mi csoportositásunk a következő:



3.2.3 Nem tipusos nyelvek (BLISS, BCPL)

A nem tipusos nyelvekben az adatstruktura egy idealizált számológép köré épül. Az adatokat a memóriában tároljuk. A memóriát kvantáltnak tételezzük fel, véges számu szóból (rekeszből) áll, a szavak pedig azonos számu bitből állnak. A rekeszeknek cime van, amely szomszédosoknál 1-gyel nő. A változók a memóriából változó hosszuságu darabokat foglalhatnak le, egy szó egészszámu többszörösét. A változó hosszát a deklarációban explicit meg kell adni, mivel nincs egyéb információ (pl. tipus),

amiből ez kiderülne. Ha a méret nincs előirva, akkor 1-nek feltételezzük. Az adatokon végzett müveletek természete és értelmezése az operátorok jellemzője. Pl. két egyszavas adatot összeadhatunk egészszámként vagy összehasonlithatunk karakterként stb. Az operátor tudja, hogy milyen hosszu adatmezőn dolgozik.

A BLISS nyelvben egy változó neve nem a változóhoz rendelt memória terület tartalmát, hanem első szavának a cimét jelenti. Ezért az indirektség kifejezésére valamilyen eszközt kell bevezetni. Erre szolgál a "vedd a tartalmát" müvelet, amelyenek "." a jele.

Ezért pl. a szokásos A=B átutalást A<--.B alakban kell irni.

Ezen szemléletmód előnye, hogy egy kifejezés kiértékelése nem függ a környezettől, pl. attól, hogy jobb vagy bal oldalon áll.

A BCPL nyelv az indirektséget másként kezeli. Megkülönböztet jobbértéket és balértéket, illetve jobb- és balmodu kiértékelést. Egy memória rekesz tartalma a jobbérték és cime a balérték, ami mindig egészszám. A balértéket is ugyanolyan hosszuságu bitminta reprezentálja, mint a jobbértéket (egy teljes szó). A kiértékelés módja a változó környezetétől függ. Értékadó utasitás bal oldalán álló kifejezést balmódon kell kiértékelni, hogy eredményként balértéket kapjunk. A bal oldalon álló kifejezés azonban csak változó név, indexes változó vagy indirekt memória hivatkozás lehet. Az ilyen kifejezéseket baltipusu kifejezéseknek hivjuk. Megjegyezzük, hogy ezek tulajdonképpen nem igazi kifejezések, a hagyományos programozási nyelvek ezeket változónak nevezik és megengedik, hogy bal oldalon álljanak.

A jobb oldalon álló kifejezés kiértékelése jobbmódon történik, azaz a változók jobbértékeit (tartalmát) kell venni és elvégezni az előirt müveleteket, igy végül is egy jobbértéket kapunk. Ily módon az értékadó utasitás (átutalás) a hagyományos módon történik. Természetesen bal oldalon csak olyan kifejezéseknek van értelme, amelyhez memóriarekesz van hozzárendelve, de erről a programozónak kell gondoskodnia.

A jobbmódu, illetve balmódu kiértékelést operátorral is előirhatjuk. Erre szolgálnak az rv. (right value), ill. lv. (left

value) operátorok, amelyek precedenciája magasabb, mint a környezeté.

Nézzük meg az rv, lv operátorok értelmezését a környezettel való kölcsönhatásban.

	rv (indirektség) felreferenciálás	lv (cimképzés) lereferenciálás
átutalás bal oldalán	rv p: = t t tartalmát tedd a p tartalma által mutatott helyre	lv p: = t nincs értelme, mert p cimét változtasd t-re lenne, de ez lehetetlen
átutalás jobb oldalán	p: = rv t t tartalmának tartalmát tedd p-be	p: = lv t t cimét tedd p-be

Természetesen lv(rv E) = E és
rv(lv E1)= E1.

Az lv operátort csak baltipusu kifejezésre lehet alkalmazni, tehát E1-nek baltipusunak kell lennie, ez biztositja a második egyenlőséget.

Mint azt 4.2.2-ben látni fogjuk, az indirektség ilyen bevezetése kényelmes indexelési és adatstruktura-kezelési lehetőséget biztosit.

Ha az adatoknak nincs tipusa, akkor természetesen tipusellenőrzés sincsen. Az adatoknak a deklarációban rögzitett másfajta tulajdonságai, pl. memóriaosztály, összetettség, azonban megvannak és explicit deklaráció is általában van a nem tipusos nyelvekben is.

3.2.4 Egyszerü önálló tipusok

Egyszerü önálló beépitett tipusok:

számok (integer, longinteger, real, longreal stb.)
karakter
boolean

A számok szempontjából a SIL nyelveknél a legnagyobb változatosság van kezdve a csak integerrel rendelkező MODULA-tól az IMP-ig, ahol byteinteger, shortinteger, integer, real és longreal tipusu számok is vannak.

Az aritmetikai operátorok precedencia sorrendben a következők:

- előjelváltás
↑ hatványozás
*, /, DIV, MOD szorzás, osztás, egészosztás, maradékképzés
+, - összeadás, kivonás

Néhány nyelvben shift müveletek és bitenkénti logikai müveletek is alkalmazhatók számokra.
Pl. XXPL, GESAL

A karakter tipusu változó fontos szerepet tölt be a SIL nyelvekben. Értékkészlete az adott gépen használatos jelkészlet, pl. ASCII vagy EBCDIC kódkészlet. Néhány nyelvben azonban csak összetett alakja, a karakterfüzér, string szerepel. Pl. IMP, XXPL. A stringet az összetett adattipusnál tárgyaljuk.

A karakter tipusu változókon végezhető müveletek az egyes nyelvekben nagyon eltérőek. A PASCAL-ban és MODULA-ban csak reláció alkalmazható rájuk, más nyelvekben a bitenkénti logikai müveletek és a shift müveletek is, sőt a +, - aritmetikai operátorok is (C-nyelv).

A boolean tipus értékkészlete a TRUE és FALSE logikai értékek. A boolean tipusu változók feltételként szolgálnak feltételes utasitásokban és feltételes kifejezésekben. Ezt a tipust több SIL nyelvben elhagyták és helyette reláció vagy numerikus értékek legalacsonyabb helyértékü bitje adja a logikai értéket (pl. 1 = TRUE, 0 = FALSE) (pl. BLISS, IMP, GESAL, XXPL). A boolean tipus összetett alakja a bits, amelyet később tárgyalunk.

A boolean tipuson végezhető müveletek a NOT, AND, OR és XOR logikai müveletek. Megjegyezzük, hogy ha egy nyelvben boolean tipus nem szerepel, logikai müveletek akkor is lehetnek és ekkor logikai értéket képviselő kifejezések közötti operátorok lesznek. A logikai kifejezések primary-je a reláció vagy zárójelbe tett logikai kifejezés.

Egyszerü önálló definiált tipusok:

enumeráció
label
procedure tipus

Enumeráció tipusu változó esetén az értékkészlet egy felsorolás elemei. Az elemeket a tipus definiálásakor kell megadni. Az elemek felsorolási sorrendje egy rendezést definiál az enumeráció elemei között.

```
Pl.   SZINEK = (PIROS, KÉK, ZÖLD)   - tipus definició
      SZINEK   A, B                 - változó deklaráció
      A: = PIROS                    - értékadó utasitás
```

Ilyen pl. a MARY set tipusa, a PASCAL scalar tipusa és a GESAL enumeráció tipusa.

A rendezésnek többnyire csak a gépi reprezentációban van szerepe, de pl. a PASCAL succ(x) és pred(x) standard függvényei megadják x követőjét, illetve megelőzőjét az enumerációban. Az enumeráció tipus olyan szempontból is rendezettnek tekintődik, hogy pl. subrange jelölhető ki belőle.

Enumeráció tipusu változóra csak reláció müvelet alkalmazható. Egyéb felhasználása, hogy set és subrange tipusok alaptipusa lehet valamint tömbök indexének és CASE utasitás szelektorának tipusa lehet.

Label tipusu változó

Néhány nyelvben (PL1, GESAL, MARY) a cimkék label tipusu konstansoknak tekintődnek. A label tipusu változókhoz értékként label tipusu konstansokból képzett kifejezések rendelhetők, majd GOTO utasitás argumentumaként szerepelhetnek. Ily módon label változók segitségével programkapcsolót tudunk megvalósitani. Ez azonban akár az ALGOL 60 switch-jével, akár más nyelvek CASE utasitásával könnyebben kiváltható.

A label tipusu változóknak mégis nagy szerepük lehet a SIL nyelvekben. Ugyanis pointer mutathat a cimkére. Ezeket REF LABEL tipusunak kell deklarálni és összetett adatstrukturákat, tömböt, rekordot stb. lehet létrehozni belőlük. Ezáltal például könnyen tudunk ugrótáblát megvalósitani.

A label tipusu változó fogalma egy nyelvben különböző mértékben, különböző kiépitettségi fokban lehet jelen. Legáltalánosabb, ha a label tipusu változókon aritmetikai müveletek végezhetők, az értékkészlet a használható cimtér, és a GOTO argumentuma kifejezés lehet. (Lásd PL1)

Kisebb kiépitettségben az értékadáson tul más müveletek nem végezhetők label tipusu változóban. A GOTO utasitás argumentuma egyetlen azonositó, és az értékkészlet csak a programban előforduló cimkék (GESAL). A legkisebb kiépitettségben még az értékadás sem végezhető a label tipuson. Ilyenkor tulajdonképpen csak konstansok tartoznak a tipushoz. Azért jó mégis tipusnak tekinteni, mert igy a cimkére mutató pointer ugyanugy deklarálható, mint a többi pointer.

Procedure tipus

Az eljárások neveit kezelhetjük ugy, mint procedure tipusu változókat, maguk az eljárások pedig procedure tipusu konstansok. Ebben az esetben a tipus definiciója tartalmazza az eljárás paramétereinek számát és tipusát és az eljárás eredményének tipusát. A procedure tipusu változókon csak egyetlen müvelet végezhető, hivni lehet őket. Egyéb felhasználásuk, hogy függvények paraméterei lehetnek, pointer mutathat rájuk, és tömbök képezhetők belőlük.

A procedure tipus bevezetésének előnyei, hogy igy a függvények procedure tipusu paraméterére is lehet tipusvizsgálatot végezni és a pointerek deklarálása egyöntetü. A MARY nyelvben a proc tipusu változókon értékadás is végezhető.

3.2.5 Egyszerü vonatkozó tipusok

subrange
set
pointer

Az egyszerü vonatkozó tipusok olyan tipusok, amelyek valamely más tipusra, az alaptipusra vonatkoztatva vannak definiálva és az alaptipussal igen szoros kapcsolatban állnak.

3.2.5.1 A subrange tipus az értékkészlet rendezettségét emeli ki. Itt az alaptipus értékkészletének egy tartományát jelöljük ki a subrange tipusu változó értékkészletéül. A subrange tipus alaptipusa az integer, karakter, boolean és enumeráció tipus lehet.

Pl. 1 .. 100
Monday .. Friday

A subrange tipus a PASCAL nyelvben szerepel először. Megjelenése nagy vitát keltett, sok probléma merült fel vele kapcsolatban. A problémák lényege az, hogy tekinthető-e a subrange tipus valóban független tipusnak, vagy csak az alaptipus rangeének. Milyen müveletek alkalmazhatók rá és a subrange tipusu kifejezés értékének kiértékelés közben is range-en belül kell-e maradni?

A vitát az okozta, hogy a PASCAL User Manual and Report [19] subrange-el kapcsolatos megfogalmazásai homályosak és pontatlanok. Pl. 6.1.3-ban a subrange definiciója nem zárja ki, hogy az alaptipus real legyen, vagy 8.1.2 és 8.1.3 ugy definiálja a *, + és - operátorokat, hogy ezek az integer, real és tetszőleges set tipusokra alkalmazhatók és a subrange tipust nem emliti. Ebből azt gondolhatja az olvasó, hogy subrange tipusu változóra airmtetikai operátor nem alkalmazható.

A PASCAL axiomatikus leirása [18] tisztázza a helyzetet. Eszerint a subrange tipusu változón minden müvelet elvégezhető, ami az alaptipuson értelmezve volt, de a müvelet eredményének tipusa az alaptipus lesz, amely aztán csak értékadáskor fog ismét subrange tipusuvá konvertálódni, ha a bal oldalon subrange tipusu változó áll.

Tehát még az a kedvező sajátság is fennáll, hogy a subrange tipusu kifejezés értékének kiértékelés közben nem kell range-on belül maradni.

Subrange tipusu változók használata index kifejezésekben, FOR utasitásban (ciklus változó kezdő és végértékét megadó kifejezések), és CASE utasitásban (szelektor kifejezés) történik.

3.2.5.2 A set tipus az enumeráció halmaz tulajdonságát emeli ki. A set tipusu változó értékkészlete egy halmaz részhalmazai. Ezt szokták ugy is kifejezni, hogy ezen változó az alaptipusnak egyszerre nemcsak egy elemét kaphatja értékül, hanem többet is. Azonban prceizebb, ha azt mondjuk, hogy a részhalmazt kapja értékül. Az alaptipus tehát mindig véges tipusu enumeráció vagy subrange, ennek részhalmazai adják a set tipusu változó értékkészletét. Az EMPTY egy speciális set tipusu érték az üres részhalmazt jelöli.

A set tipusu változókon halmaz müveletek végezhetők, természetesen csak azonos alaptipushoz tartozó setek között. Ilyen müveletek:

+ union, egyesités
- set difference, halmazok különbsége
* intersection, metszet
IN membership, tartalmazás

A membership müvelet eredménye boolean tipusu.

Egy set tipusu érték felfogható logikai értékek, bitek sorozatának. A megfeleltetést ugy végezzük, hogy az alaphalmaz elemeihez rendre egy-egy bitet rendelünk, és egy aktuális értékben azoknak és csak azoknak a biteknek az értéke = 1, amelyek beletartoznak az értékül vett részhalmazba. Ily módon az union halmazmüvelet equivalens a bitenkénti logikai OR müvelettel, az intersection pedig a bitenkénti logikai AND-del. ezek mintájára az XOR logikai müveletet is bevezethetjük A XOR B ≡ (A OR B) - - (A AND B) definicióval.

Megjegyezzük, hogy a set tipus némely nyelvben (pl. GESAL) nem származtatott, hanem önálló tipus, hasonlóan, mint az enumeráció. A különbség az, hogy a set tipus nem egy már definiált enumerációra vonatkozik, hanem a set tipus definiálásakor kell

megadni az alaphalmaz elemeit. Az eltérés szemléltetésére megadjuk egy set tipusu változó definicióját PASCAL-ban és GESAL-ban.

```
PASCAL:    COLOR = (RED, YELLOW, GREEN, BLUE)
           S1: SET OF COLOR
GESAL:     SET (PIROS, KÉK, ZÖLD)  S1;
```

3.2.5.3 Pointer tipusu változó

A pointer változó értéke valamilyen adatra mutató pointer. A mutatott adat tetszőleges tipusu lehet, de tipusát deklaráláskor rögziteni kell. Csak az azonos tipusra mutató pointer változók helyettesithetők egymással. A pointer tipust általában a REF kulcsszó, más nyelvekben ↑ vagy * jelöli. Használata kétféle lehet.

Statikusan használjuk őket paraméteradásnál. Ugyanis, ha tömböt vagy rekordot akarunk átadni, akkor az adat átmásolása helyett célszerü csak az adatra mutató pointert átadni. Akkor is statikus használatról van szó, ha azonos memória területre való többféle hivatkozást valósitunk meg pointer változókkal vagy az algoritmust gyorsitjuk meg az adatok cimeinek közvetlen meghatározásával.

Másik használatuk a dinamikus változók, illetve a dinamikusan kezelt adatterületek használatával kapcsolatos. Ezeket nem a nevükkel, hanem csak a rájuk mutató pointerrel azonositjuk. Kezelésük standard eljárásokkal történik. Generálásuk a new eljárás segitségével valósul meg.
Pl. p = ref tl tipus esetén new(p) területet foglal egy tl tipusu cella számára és a visszatérő érték egy, a foglalt cellára mutató pointer, amely értékadással egy p tipusu változónak adható. A dinamikusan kezelt adatterület felszabaditása is standard eljáráson keresztül valósul meg (dispose). A pointer tipusu változókon végezhető müvelet első kiépitettségben az értékadás és az egyenlőség reláció (PASCAL, GESAL), természetesen azonos tipusu változókra mutató pointerek között. Nagyobb kiépitettségben összeadás, kivonás és reláció müveletek is végezhetők, pl. C-nyelv. A NIL egy speciális pointer tipusu érték, azt mutatja, hogy a pointer semmilyen elemre sem mutat.

3.2.6 Összetett adattipusok

Önálló összetett adattipusok

Azonos tipusu elemekből összetett adatsokaság a tömb és a file, különböző tipusu elemekből felépülő adatstruktura pedig a rekord.

3.2.6.1 A tömb rögzitett méretü, direkt elérésü rendezett sokaság. A méret rögzitése történhet a fordítás idején (fix méretü tömbök, pl. FORTRAN, PASCAL, GESAL), történhet egy blokkba való belépéskor (változó méretü tömb, pl. ALGOL-60) vagy blokkon belül is (flexibilis tömb PL/1, ALGOL-68). A rendezés az indexek értékei szerint történik. Az indexek száma a tömb dimenziója, ez több nyelvnél korlátozva van. Az alaptipus általában tetszőleges már definiált tipus lehet, az indexek tipusa általában integer, de némely nyelv megenged más tipust is, pl. a PASCAL tetszőleges enumerációt és subrange-t.

A tömbstruktura definiálása általában az alaptipus és az alsó és felső indexhatárok megadásával történik.

Pl. alma array of boolean (a_1: f_1, a_2: f_2, ...). Az alsó indexhatár elmaradása esetén azt megállapodás szerint O-nak vagy 1-nek feltételezzük. Az indexek tipusát általában az indexhatárokat megadó kifejezés tipusa határozza meg. A tömbön mint egészen az átutaláson kivül más müvelet nem végezhető, ez sincs minden nyelvben megengedve.

Csak azonos méretü és alaptipusu tömbök cserélhetők fel. A tömb elemeire az indexelt változók segitségével hivatkozhatunk. Indexelésnél a beépitett elérési mechanizmus a következő:

Ha A : array of B [k_1, k_2, ..., k_n]
akkor az A [i_1, i_2, ..., i_n] indexelt változó elérése

addr (A) + b*(...((i_1-1)*k_2 + i_2-1)*k_3 + ,,, i_n - 1)

amennyiben az indexek közül i_n változik a leggyorsabban; itt
addr (A) az A tömb kezdőcime,
b egy B tipusu elem elhelyezéséhez szükséges memóriatartomány hossza,
i_1, i_2... az indexek aktuális értékei és
k_1, k_2,... az index tartomány számossága.

Egy index aktuális értékét meghatározó index kifejezés tipusának meg kell egyeznie a deklarációban szereplő indextipussal.

Változó méretü tömbök esetén a fordító csak a tömbleiró elemet készíti el. Az indexelt változó cimének kiszámitása futáskor történik. Ez a tárgykód hatékonyságát nagyon rontja. Ezért sok SIL nyelvben nincs dinamikus indexhatár. Pl. GESAL, MODULA. Flexibilis tömb hatékonysági megfontolásból szintén egyáltalán nem szerepel SIL nyelvekben.

3.2.6.2 A file szekvenciális elérésü, azonos tipusu elemekből álló adatsokaság. A file-nak nincs rögzitett hossza, azaz a benne szereplő elemi adatok száma változhat. A file-nak mindig csak egy eleméhez férünk hozzá, amelyet az aktuális file pozició (iró/olvasó fej) határoz meg. Az adatcsere egy egyszavas bufferen, az un. buffer változón keresztül történik. Deklaráláskor a file-néven kivül az alaptipust kell megadni. Egy file tipusu változó deklarálásakor járulékosan a buffer változó is deklarálódik, ami egy, az alaptipusra mutató referencia tipusu változó és jelölése: file név ↑.

A file-okkal való adatcserét beépitett eljárások bonyolitják le.

Ezek:

put(f) Az f↑ buffer változó értékét hozzáfüzi az f filehoz. Ezt a müveletet csak a file végén lehet elvégezni (az eof predikátum = true).

get(f) Az aktuális file poziciót a következő elemre állitja és ezen elem értékét az f↑ buffer változóhoz rendeli értékül. Ha nincs következő elem, akkor az eof predikátum true-vá válik és f↑ értéke definiálatlan. A get(f) müvelet csak eof = false esetén van definiálva.

reset(f) Ha f nem üres, akkor visszaállitja az aktuális file poziciót a kezdőpontra és az f↑ buffer változóhoz f első elemét rendeli értékül, egyidejüleg eof = false lesz. Ha f üres, akkor eof = true marad, az aktuális file pozició a kezdőpontra mutat és f↑ értéke definiálatlan.

rewrite(f) f jelenlegi értékét eldobja, ugyanarra a területre uj file irása kezdődhet, eof = true lesz.

A tárgyalt nyelvek közül csak a PASCAL-ban és a TARTAN-ban van file.

3.2.6.3 A rekord különböző alaptipusu elemekből összetett adatstruktura. Egy rekord rögzitett számu és tipusu komponensekből, un. mezőből áll. A rekord definiciója minden mezőhöz megadja a nevét és a tipusát. A mezőnevek hatásköre csak a rekord definicióra terjed ki, más rekord definicióban szintén használhatjuk ugyanazokat a neveket. Kivétel pl. a C-nyelv.

```
Pl. A: rekord     name, firstname: string;
                  age: 0..99;
                  married boolean;
       end
```

Több SIL nyelvben lehetőség van rekurziv adatstruktura létrehozására. Ez alatt azt értjük, hogy egy rekord tartalmazhat saját tipusára mutató pointert vagy két rekord kölcsönösen tartalmaz a másikra mutató pointert. Ez a sajátság nagyon hasznos pl. listák és fák kezelésénél. Az természetesen nem képzelhető el, hogy egy rekord a belsejében saját tipusu rekordot tartalmazzon, mert ekkor helyigénye meghatározhatatlan. Mivel a pointer mindig ugyanakkora helyet foglal, bármilyen tipusra mutat is, ezért egy rekord helyfoglalása egyértelmüen meghatározható akkor is, ha a benne szereplő pointerek által hivatkozott tipus helyfoglalása még nem ismeretes. A megvalósitás módja az egyes nyelvekben némileg eltérő. A következő variációk vannak:

- Csak már definiált tipusnév hivatkozható. Ez esetben rekurziv adatstruktura nem lehetséges.
- Rekordon belül előfordulhat saját maga tipusára vonatkozó referencia.
  Pl. PASCAL, C-nyelv.
- Bizonyos poziciókban megengedünk postdefiniálást (GESAL-ban REF alaptipusaként, IMP-ben rekordmezőként álló rekordra mutató pointernél).

- A tipusnév definiálás és tipus leirás szétválása (MARY, ADA).

  Pl. MODE NODE
      NODE = ...

  Az utóbbi két esetben kölcsönös rekurziót is létre lehet hozni.

A változó rekord fogalma. Változó rekord esetén egy rekord tipusnak több variánsa lehet. (PASCAL, TARTAN stb.) Ebben az esetben a rekord egy speciális mezőt, un. szelektort (tag field vagy discriminant component) tartalmaz, amelynek aktuális értéke mutatja, hogy az adott hivatkozásban melyik variánst kell venni. Minden variánst egy cimke azonosit, amely a szelektor tipusával azonos tipusu konstans.

```
Pl.  NEM = (FÉRFI, NOE)
     RECORD SZEMÉLYIG: INT;
            MUNKAKÖNYV: INT;
     CASE S: NEM OF
     FÉRFI: (KATONAIG : INT;
             HÁZAS: BOOLEAN)
     NOE  : (SZÜLÉSEK SZÁMA : INT)
     END
```

A szelektor csak az egész rekorddal együtt irható.

Egy rekord mezőire az un. minősitett névvel hivatkozhatunk, ami a konkrét rekord nevéből, azt követően egy pontból és egy mezőnévből áll.

```
Pl. A.name
    p2↑.side
```

Változó rekord esetén az aktuális variáns mezőneveivel minősitünk.

Vonatkozó összetett adattipusok

bits
string

3.2.6.4 A bits adattipus az array of boolean rövid neve. A bits tipus általában rögzitett hosszuságu (byte, szó) bitsorozatokat tartalmaz (pl. MODULA), de van változó hosszúságu változata is

(pl. XXPL). A bits tipus leglényegesebb eltérése a set tipustól az, hogy a setnél az egyes bitek meg vannak nevezve, mig a bitsnél nem, emellett a set hosszuságát mindig a felhasználó definiálja.

A bitkonstansban azokat a biteket vagy bit tartományokat adjuk meg, amelyek értéke true.
Pl. [3,6:8,10] jelöli a 0010011101 bitsorozatot
(1 = ture, 0 = false).

Bits tipusu változókon a következő müveletek vannak értelmezve:

OR, XOR, AND és NOT,
azaz a szokásos bitenkénti logikai müveletek.

3.2.6.5 A string az array of character rövid neve. A string hosszát, azaz karaktereinek számát a string tipusu változó deklarálásakor kell rögziteni vagy a kezdőértékként kapott string hossza dönti el. A stringgel mint egységgel különféle müveleteket lehet végrehajtani, de a karaktereihez is hozzáférhetünk.

A string müveletek egy részét standard függvények végzik.

Karakterkiemelés: byte (v, u) a v string u-edik karakterét teszi hozzáférhetővé.

hossz-meghatározás: length(v) a v string hosszát adja meg.

szelet: substr (v, u, ℓ) a v string u-edik karakterével kezdődő, ℓ hosszuságu karaktersorozatát adja eredményül.

farok: substr (v, u) a v stringből elhagyja az első u karaktert.

relációk: =, $\neq$, >,<,= <,> =
Két string akkor és csak akkor egyenlő, ha mind hosszuk, mind karakterenkénti értékük megegyezik.
A string > B string, ha
length(A) > length(B) vagy
length(A) = length(B), de az első eltérő karakter A-ban a nagyobb.

Müveletek: konkatenáció, egymás után füzés: Egy n és egy m hosszuságu string egymás után füzése egy n + m hosszuságu stringet eredményez.

resolution: L → M.(E).N
itt L, M és N string változók
és E string kifejezés.
A resolution értéke = true, ha az E string kifejezés értéke az L string egy szelete. Ez esetben az M változó értékül kapja L kezdőstringjét E legbaloldalibb előfordulásáig, N pedig az E legbaloldalibb előfordulása utáni farokrészt.
Ha E nem szelete L-nek, akkor a resolution értéke = false és értékadás nem történik. Ez a müvelet csak az IMP nyelvben található.

3.2.7 Union tipus

Az union tipus lehetőséget teremt arra, hogy ugyanahhoz az adathoz különböző uton hozzáférhessünk. Szemantikájában a FORTRAN nyelv EQUIVALENCE utasitásának felel meg.

A GESAL és a C-nyelv union tipusa ezt a célt szolgálja. Az ugyanarra a memória területre foglalandó tipusokat az union tipus deklarációjában kell felsorolni. Egy union tipusunak deklarált változó számára foglalt memória hely az union deklarációban szereplő leghosszabb elem hossza.

Az union tipus használata pl. a GESAL nyelvben a következőképpen történik.

MODE typename = UNION (type1 name1; type2 name2; ...);
Ehhez a tipushoz egy változó deklarálás:
typename varname;
A hivatkozás minősitett névvel történik:
varname.name1
Példa: MODE UNI = UNION (REAL R; [2] INT I);
UNI U; INT I;
U. I[0] → I;

Hasonló célt szolgál a PL360 szinonim deklarációja és a CORAL-66 nyelv overlay tipusa is.

3.2.8 A tipusdefiniálás problémái

A definició és deklaráció általában nem válik szét élesen. Például tömböt a legtöbb SIL nyelvben deklarálhatunk ugy, hogy definiáljuk az adott tömbtipust és nevet adunk neki, majd deklarálunk egy változót ehhez a tipushoz tartozóan.

Pl. A = TYPE ARRAY OF T1 $[a_1 : f_1]$
A ALMA;

Az ALMA tömb deklarálását azonban végezhetjük közvetlenül is, igy:

ARRAY of T1 $[a_1 : f_1]$ ALMA

Ez a lehetőség a PASCAL-ban, IMP-ben, MODULA-ban is megvan.

Hogy a kétféle deklarálás azonos értékü-e, az a nyelv szemléletétől függ. Az uj tipusnevek bevezetése ugyanis felveti a tipusok azonositásának problémáját, ami a következő: azonosnak számit-e két tipus, ha definiciója azonos, de neve különböző. Az egyik szemlélet szerint a tipusnév csak röviditésnek számit és helyette mindig a kifejtett tipusdefinciót kell gondolnunk. Tehát a különböző nevü tipusok azonosak, ha kifejtésük azonos. Ezzel azonban elvesztünk egy csomó előnyt, ami a tipusellenőrzéssel járna.

Példa:

```
TERÜLET = REAL     }
HOSSZUSÁG = REAL   }   tipusdefinició
HOSSZUSÁG A, M, M1, M2
TERÜLET T, T1, T2
```

A háromszög területe: T1 = A * M1/2
T2 = A * M2/2
M = M1 + M2
T = T1 + T2 = A * (M1 + M2)/2 = A * M/2

Ebben a példában minden mennyiség real. Ha a végzett tipusdefiniciókat röviditésként értelmezzük, akkor semmiféle előnyhöz nem jutunk. Ellenben, ha a más névvel nevezett tipus másnak számit, és közöttük nem végezhető müvelet, akkor tipusellenőrzéssel kiszürhetők a T + M és hozzá hasonló hibás kifejezések.

Ebben az esetben szükség van arra, hogy a tipussal együtt a rajta végezhető müveleteket is előirjuk.

Tulajdonképpen arról van szó, hogy a hagyományos nyelvek tipusfogalmába három egymástól független sajátság van összevonva. Ezek:

az értékkészlet

a végezhető müveletek és

az összetettség.

Az értékkészlet szükitésére a subrange tipus szolgál, ekkor a végezhető müveletek köre nem csökken (lásd subrange tipus 3.2.5-ben).

Kisérletek történtek a müveletek korlátozására az értelmezési tartomány változatlanul hagyása mellett. Ilyen kisérlet például Cleavelandé [29], aki un. "pouch"-okat vezet be azonos tipusu, de mégis különbözőképpen kezelendő mennyiségek számára. Cleaveland maga "dublikált" tipusról beszél, ami tulajdonképpen az eredeti tipusfogalom kettéválása. Pouch-ok bevezetésével a dimenzió elemzéshez hasonló elemzést tudunk elvégezni a szereplő mennyiségeken forditási idő alatt.

A tipusdefiniálással kapcsolatos problémák megnyugtató megoldása csak az ADA nyelvben történt meg. Mivel e tanulmány irásakor az ADA nyelv még nem létezett, ezért nem is szerepel az ismertetett nyelvek között. Azonban a kiadás elhuzódott és a közben megjelent ADA meghozta a tipusproblémák megoldását. Ezért ezt itt ismertetjük anélkül, hogy a teljes ADA nyelv ismertetését megtennénk. [30]

Az ADA egyébként a Steelman követelményeit kielégitő, elfogadott programozási nyelv.

Az ADA fő érdeme, hogy eszközt biztosit arra, hogy a programozó maga irja elő a származtatott tipusnak az eredetihez való kapcsolatát. A tipusokat származásuk szerint bizonyos hierarchiába rendezi. Megkülönbözteti a tipus, az altipus (subtype) és a származék tipus (derived type) fogalmát.

A tipus megadja a lehetséges értékek készletét és a rajta végezhető müveleteket. Lehet beépitett tipus pl. INTEGER vagy felhasználó definiálta tipus, pl enumeráció.

Az altipus a bázis tipus értékkészletét szükiti le korlátozás bevezetésével. Nem uj tipus. Minden operátor és literál közös a báziséval. A PASCAL nyelv subrange fogalmának felel meg.

A származék tipus adatstrukturája azonos a bázistipuséval, az értékkészletet, literálokat, operátorokat örökli a bázistól. Ezzel szemben uj tipus, a bázistipussal csak konvertálás után irható egy kifejezésbe. Őseivel oda-vissza konvertálható. Az öröklött operátorok felülirhatók.

A tipusoknak ez a származási hierarchiája független a tipusok adatstruktura szerinti megkülönböztetésétől (enumeráció, numerikus tipus, tömb, rekord és referencia tipusok). A korlátozás viszont adatstrukturáktól függ. Egy deklarációban a kétféle származtatás összevonható és pl. származék tipus altipusát hozhatjuk létre.

Példa:

tipus definició:

```
type WEEK is (MON, TUE, WED, THU, FRI, SAT, SUN);
```

altipus definició:

```
type WEEK1 is new WEEK
```

kombinált defincó:

```
type WORKDAY is new WEEK (MON..FRI);
```

Az utolsó esetben létrejön egy névtelen származák tipus, amelynek jellemzői teljesen megegyeznek a báziséval, de mégis külön tipus. Ennek a névtelen származéktipusnak az altipusa lesz a WORKDAY nevü tipus.

3.2.9 Tárkezelés

Tárkezelés szempontjából a memória osztály és a heap memória fogalmát tárgyaljuk.

A memória osztály mint a változó egy attributuma, általában prefix, utasitást ad a változó elhelyezésére vonatkozóan. Általában a következő memória osztályok vannak:

A globális változó a program futása során végig él, minden blokkból és eljárásból hivatkozható. Helyfoglalása a statikus memória területen történik.

Az external változónak nem foglalódik hely, mert annak helye valamely másik szegmensben van. Különben globálisként viselkedik.

A statikus változó az ALGOL-60 own tipusu változójához hasonlóan viselkedik. Helyfoglalása a statikus memória területen történik. A globális változótól annyiban tér el, hogy mig a globális változó mindenhonnan, addig a statikus változó csak saját blokkjából hivatkozható.

Lokális változó, amely csak a blokk vagy eljárás belsejében él, kivülről nem hivatkozható és helyfoglalása a blokk számára a veremben foglalt területen történik.

A regiszter-ként kijelölt változó helyfoglalása nagyobb sebességü memória területen (hardware regiszterek) történik.

Heap memória. A pointer változó bevezetése a stack-szervezésü memóriakezeléssel rendelkező nyelvekben a következő problémát veti fel. Abban az esetben, ha egy pointer változó olyan változóra mutat, amelynek hatásköre kisebb, mint a pointeré, akkor a mutatott változó hatásköréből kilépve a pointer értéke meghatározatlan lesz, mivel a kilépett blokk helyét fel kell szabaditani.

Ezen probléma egzakt megoldása a heap tipusu tárkezelés bevezetése. Ennek az a lényege, hogy a kilépett változók helyét nem szabaditjuk fel addig, amig pointer mutat rájuk, vagy amig külön utasitással nem töröljük őket. Ennek megvalósitására két módszer kinálkozik. Az egyik - és ez a szokásosabb - az, hogy a kilépett programegység helyét nem szabaditjuk fel, az uj egység számára a még fel nem használt területből foglalunk helyet. Amikor a szabad terület elfogy, akkor a garbage collector kigyüjti és tömöriti az élő hivatkozásokat, a nem hivatkozott változók helyét pedig felszabaditja. A másik lehetséges módszer, hogy minden változóhoz referenciaszámlálót rendelünk, amely számlálja a rá mutató pointereket. Amennyiben a referenciaszámláló 0-vá válik, akkor a helyét felszabaditjuk.

A heap tipusu tárkezelés mindkét megvalósitása nagyon lelassitja a program futását, ezért tiszta heap tárkezeléssel nagyon kevés helyen találkozunk.

A fenti probléma megoldása szempontjából a következő esetek vannak:

a/ Csak stack kezelésü memória van. Ebben az esetben ujabb két eset lehetséges. A nyelv statikus szemantikája megtilthatja, hogy egy pointer nála kisebb hatáskörü változóra mutasson. Ennek compile-time felderitése azonban nem könnyü feladat a compiler számára. Némely nyelv megengedi a pointer tetszőleges használatát is, de ekkor a programozónak nagyon kell ügyelni erre, mert nagy a hibalehetőség.

b/ A nyelvben van verem kezelésü és heap kezelésü memória is. Pl. PASCAL, MARY, TARTAN. A PASCAL-ban és TARTAN-ban a heap a dinamikus változók tárolására szolgál. (Lásd 3.2.5.3) Ezek mereven elkülönülnek az egyéb, nem dinamikus változóktól, amelyek veremben vannak. A PASCAL-ban egy pointer kizárólag dinamikus változóra mutathat.

A MARY-ben tulajdonképpen nincs kiépitett heap, de a cella, terület és generátor deklarálási lehetőség eszközt ad arra, hogy a programozó maga döntse el minden változójáról, hogy hogyan kivánja kezelni (lásd 3.2.10.2).

c/ Csak heap van. Ilyen nyelv pl. CLU. A SIL nyelvek között nincs ilyen.

3.2.10 Deklarációk

A deklarációkkal kapcsolatban a következő dolgokról kell beszélni:

- a deklarációk helye és kényszere
- a deklarációk tartalma
- kezdőértékadás.

Itt a deklarációkba a definiciókat is beleértjük.

3.2.10.1 A deklarációk helye

1. A program elején kell állniok. Pl. IMP, PASCAL, MODULA, BCPL.
2. A deklaráció bárhol állhat, de meg kell előznie a hivatkozást. Pl. XXPL, GESAL, MARY.

3. Bárhol állhat deklaráció, de kell, hogy legyen.
4. Nem kötelező a deklaráció. Pl. C-nyelv. Implicit deklaráció esetén a változó tipusát a szövegkörnyezet határozza meg (pl. a név kezdőbetüje, kifejezés tipusa stb.). Más esetekben a nem deklarált változó tipusa valamilyen default értéket kap, pl. integer. A deklaráció kényszer a forditás technikájával van szoros kapcsolatban. Egymenetes forditás esetén nyilvánvalóan szükségszerü, hogy a deklaráció megelőzze a hivatkozást. Más a helyzet a postdefinit tipus definiciónál. Ezt a 3.2.6.3-ban tárgyaltuk.

3.2.10.2 A deklarációk tartalma

Deklarálni kell általában a program különböző épitőelemeit. A legfontosabbak: literál, konstans, tipus, változó, eljárás.

A literál definició egy szövegrészhez nevet rendel. Ahol a programban a név előfordul, ott helyette mindenütt a szövegrész helyettesitődik.

A konstans deklaráció összerendel egy nevet és egy tipussal rendelkező értéket. A tipust az érték külalakja határozza meg. A név a programban az adott tipusu read-only változóként viselkedik.

A tipus definició nevet ad egy definiált tipusnak. Helyfoglalás nem történik.

A változó deklaráció lefoglal az adott tipusu változó tárolásához szükséges helyet az attributum által mutatott memória területen.

Az eljárás deklaráció megadja az eljárás nevét és a paramétereinek számát és tipusát, függvény esetén a visszatérő paraméter tipusát. Az eljárás törzsének leirása általában szintén beletartozik az eljárás deklarációba. Azonban van olyan nyelv, pl. IMP, ahol a törzs leirása különválhat az eljárás deklarációtól.

A legtöbb SIL nyelv csak a fenti objektumok deklarálását irja elő, illetve teszi lehetővé. Ezenkivül egy-egy nyelvnél előfordulnak a következő deklarációk: cimke, cella, terület, generátor, operátor, kényszeritès és kontext.

A cimkék deklarálásának előirása általában a használatuk megnehezitését célozza. A nyelvek szerzői ilyen módon próbálják a felhasználókat cimkék és GOTO nélküli strukturált programok irására ösztönözni. Pl. MARY, PASCAL. A cimke deklarációról bővebben 4.4.1.2-ben hallhatunk.

Cella, terület és generátor deklaráció a MARY nyelvben fordul elő. Itt a felhasználó közvetlenül szabályozhatja a tárkiosztást. A cella deklaráció megadja a cellában tárolandó adat tipusát, módját, helyet foglal számára és esetleg kezdeti értékkel is feltölti. Hasonlit más nyelvek változó deklarációjához. A celladeklarációk (és az eljáráshivások) a szükséges tárigényt un. területekből kapják, amelyek generátorok segitségével jönnek létre. Két standard generátor van: a "RECURS" generátor, amely a rekurziót lehetővé tevő veremszervezést teszi lehetővé, és a "GLOBAL" generátor, amely FORTRAN-szerü statikus helykezelést valósit meg.

Operátor, kényszerités és kontext deklarációk. Az operátorokat a MARY-ben deklarálni kell. Az operátor deklarációban makrotörzsként kell megadni az elvégzendő müvelet leirását. Ugyanazzal az operátorral (pl. +) több müvelet is jelölhető, pl. egész-összeadás, valós-összeadás, ekkor az operandusok tipusa dönti el, hogy melyik az alkalmazandó müvelet. A legfontosabb operátorok deklarációját a standard prelude és az implementációs prelude-ök tartalmazzák.

Ha egy adott környezetben egy adott tipusu érték nem szerepelhet, akkor a forditóprogram megkisérli az értéket a kivánt tipusuvá kényszeriteni. A kényszeritések a MARY-ben deklarálhatók, számos kényszerités a standard prelude-ben van deklarálva.

A kényszerités deklarációban egy paramétert és az eredmény tipusát kell specifikálni. Mind az operátor, mind a kényszerités deklarációban szerepe van a kontextnek. A kontext a kényszeritések egy csoportja, amelyet a compiler az adott körülmények között használhat. A kontext deklaráció kényszeritések és más kontextek felsorolását tartalmazza.

```
Példa: COERCION NARROWING, WIDENING;
       CONTEXT NARCON = (STRONG, NARROWING),
               WIDCON = (STRONG, WIDENING),
               STRONG = (STRONG, WIDENING, NARROWING);
       COERCION NARROWING (VAL REAL A NARCON)VAL INT =
                  (A FIX) $,
               WIDENING (VAL INT A WIDCON)VAL REAL =
                  (A FLOAT)$;
```

Ezek a deklarációk bevezetik az automatikus konverziót REAL és INT között és felül deklarálják a standard prelude STRONG kontextjét olymódon, hogy a konverzió automatikusan megtörténjen, ha mind az apriori, mind az aposteriori tipus ismert.
Pl. REAL R:=5.
NARROWING = szükités, egy REAL számot INT-té konvertál a FIX rutin alkalmazásával, mig a WIDENING egy INT számot szélesit REAL-lé (FLOAT rutin).

3.2.10.3.Kezdőértékadás

A legtöbb SIL nyelvben a változóknak deklaráláskor kezdőértéket adhatunk, ami a változó tipusával azonos vagy abba automatikusan konvertálható konstans.
Legáltalánosabb formája:
T1 N1 = K1, N2 = K2, ... ;
vagy
T2 NS = (E1, E2, ..., K(EL, ...), ...)
Itt T1 egyszerü, T2 összetett tipus, N1, N2 változónevek, NS összetett adatstruktura neve, K1, K2, E1, E2, EL megfelelő tipusu konstans kifejezések, K pedig egész szám, az ismétlési tényező.

Tömbök és rekordok esetén a kezdeti értékek beállitása zárójelbe tett listával történik. A lista elemei vesszővel vannak elválasztva és lehetnek ismétlési tényezővel ellátott zárójelbe tett listák. A kezdőértékek tipusának természetesen meg kell egyezni a struktura által megkivánt tipusokkal. Rekordok esetén általában a rekord mezőinek megfelelően kell zárójelezni. Ha a lista a kivántnál kevesebb elemet tartalmaz, akkor a maradék elemek definiálatlanok maradnak vagy zérussal töltődnek fel.

Implicit tömbméret meghatározás esetén a tömb méretét a kezdeti értékadással határozzuk meg. Ilyenkor a deklarációban utalás van erre, pl. [] vagy [*] áll a méretmeghatározás helyén.

Példák: REAL PI = 3.14, E = 2.71;

ARRAY OF INTEGER [9] A = (1, 2, 3, 2(4, 5, 6))

ARRAY OF CHAR [] TEXT = ('PROGRAM')

Az utóbbi deklaráció a TEXT tömb méretét 7-nek definiálja.

A kezdőértékadás különböző nyelvekben különböző kiépitettségben van meg az alábbiak szerint:

a/ milyen memória osztályu változóknak lehet kezdőértéket adni?
- semminek (PASCAL)
- csak azoknak, amelyeknek a statikus memória területen foglalódik hely és nem a veremben (global, static memória osztályok). (IMP, GESAL, C-nyelv)
- nincs megszoritás (PL360, XXPL), azonban ezekben a nyelvekben nincs igazi rekurzió, tehát veremmemória nincs, igy ez a csoport lényegében azonos az előzővel
- nem statikus változó is kaphat kezdőértéket, BCPL.

b/ A kezdőérték konstans vagy konstans kifejezés lehet.
- konstans kifejezés (BCPL, C-nyelv, GESAL)
- konstans (PL360, XXPL, IMP).

c/ Tömb kaphat-e kezdőértéket?
- nem (BCPL)
- csak fix méretü egyszerü adattipusból felépitett egydimenziós tömbök. (PL360, XXPL, IMP)
- minden tömb kaphat kezdőértéket (GESAL)

d/ Implicit tömbméret meghatározás
- van: PL360, XXPL, GESAL, C-nyelv
- nincs: BCPL, IMP

e/ Ismétlési tényező
- van: PL360, IMP
- nincs: XXPL, GESAL, C-nyelv

f/ Rekord kaphat-e kezdőértéket?
- kaphat: GESAL, C-nyelv
- nem: IMP

3.3 Vezérlési szerkezetek

A SIL nyelvek vezérlési szerkezetei nem különböznek lényegesen az általános rendeltetésü programozási nyelvek vezérlési szerkezeteitől. A SIL nyelveknél is előtérbe került a strukturált programozásra való igény. Ennek megfelelően a feltételes, a ciklus és az esetszétválasztó vezérlési szerkezetek kerültek előtérbe, de a SIL nyelvek többsége megtartotta a feltétel nélküli vezérlésátadást is. Egyes SIL nyelvek viszont az un. "szökés-kifejezésekkel" pótolják a kitiltott GOTO utasitást. A vezérlésátadó szerkezetek között beszélnünk kell a rutinhivás és a korutin hivás mechanizmusáról is.

3.3.1 Feltételes vezérlésátadás

A feltételes vezérlésátadás hagyományos formái
az IF F1 THEN U1;
és az IF F1 THEN U1 ELSE U2;
utasitások.

Itt F1 feltétel, U1 és U2 pedig utasitások.

A feltételes utasitás egyik problémája az, hogy hogyan jelezzük a végrehajtandó utasitássorozat végét (lásd 3.1.2). Ebből a szempontból különböző variációk vannak. Ha záró kulcsszót vezetünk be, akkor ez jelzi az U1 és U2 utasitássorozat végét. Pl. GESAL-ban a FI kulcsszó. A legtöbb SIL nyelvben U1 és U2 egyetlen összetett utasitás a BEGIN ... END kulcsszavakkal.

A feltételes utasitások legszélesebb választékával az IMP nyelvben találkozunk, ahol az U1 és U2 utasitássorozatot a %START ... %FINISH kulcsszópár fogja össze. Ennek előnye, hogy az összetett utasitás formailag is különbözik a blokktól (lásd 3.1.2.a). Ugyanakkor az egyes utasitásokat az %AND kulcsszóval is összekapcsolhatjuk egyetlen utasitássá.

```
Pl. %IF   A = 1   %THEN  %START
          NEWPAGE
          LINE = 0
          %FINISH;
```

Ugyanez másképpen;

```
%IF   A = 1  %THEN  NEWPAGE  %AND  LINE = 0;
```

Amennyiben az utasitásrész csak egyetlen egyszerü utasitást tartalmaz, akkor a feltétel elé hozható.

```
Pl.   A = 0  %IF  A = B;
```

Ha az IF kulcsszó helyett az UNLESS kulcsszót használjuk, akkor a feltétel hamis volta esetén hajtódik végre a kijelölt utasitás(sorozat). Pl. IMP, BCPL.

A feltételes utasitás egy másik problémája az IF ... THEN IF ... THEN ... ELSE ... tipusu utasitás kétértelmüségének feloldása. Mint ismeretes, az ALGOL 60-ban megtiltották, hogy THEN után IF következzen. Hasonló kétértelmüség fellépése miatt ELSE-sel rendelkező THEN után nem szabad FOR-t irni. A SIL nyelvek egy részében ezt a módszert követték. Az ELSE-sel rendelkező THEN után megtiltják vezérlésátadó szerkezet irását (PL360) vagy minden THEN után megtiltják ezt (IMP). Más nyelvek a kétértelmü esetekben az egyik értelmezést mondják ki helyesnek, például hogy a legmélyebben fekvő ELSE mindig a hozzá legközelebb álló THEN-hez tartozik (PASCAL, XXPL, C-nyelv).

A láncolt feltételes utasitások irásának röviditésére szolgál az ELSIF kifejezés.

```
IF   F1   THEN   U1
ELSIF   F2   THEN  U2
  :
ELSIF   Fn   THEN  Un
ELSE   Un+1;
```

Szemantikája:

```
IF   F1   THEN   U1   ELSE
IF   F2   THEN   U2   ELSE
  :
. IF   Fn   THEN   Un   ELSE
Un+1;
```

A feltételes utasitásban feltételként logikai kifejezés szerepel. Ha boolean tipusu változó nincs is a nyelvben, relációkból akkor is lehet logikai kifejezést felépiteni (lásd 3.2.4 boolean tipus).

3.3.2 Ciklus utasitások

A ciklus utasitásokat általánosan három tipusra oszthatjuk.

a/ feltételhez kötött ciklus

b/ ciklusváltozóval és szabályos ismétléssel

c/ ciklusváltozóval és szabálytalan ismétléssel.

Ezek azonban össze is fonódhatnak. Például az ALGOL 60 általános ciklus utasitása mindhárom tipust tartalmazza. A SIL nyelveknél az a/ és b/ tipus a döntő, ezekre általában külön utasitás van, mig a c/ tipus a SIL nyelvekben egyáltalán nem szerepel.

Feltételhez kötött ciklus (WHILE)

A feltételhez kötött ciklus esetén a ciklus befejezése feltétel fennállásához van kötve.

Legegyszerübb alakja:

```
WHILE   F1   DO   U1.
```

Itt F1 feltétel és U1 utasitás. Ennek a szerkezetnek is különböző variánsai vannak aszerint, hogy

- az U1 utasitás összetett utasitás vagy zárókulcsszóval lezárt utasitássorozat,
- a feltétel vizsgálata a ciklustörzs lefutása előtt vagy után, esetleg közben történik,
- a ciklus befejezése az F1 feltétel igaz vagy hamis voltának következménye.

a/ A ciklus lezárása szempontjából hasonló a helyzet, mint az IF utasitásnál volt. U1 összetett utasitás a legtöbb SIL nyelvben (PL360, PASCAL, BLISS, C-nyelv, BCPL). Tulajdonképpen összetett utasitás az IMP-ben is, de a használt kulcsszópár (%CYCLE ... %REPEAT) eltér az IF utasitásban használttól. Ugyanakkor itt is lehetőség van rá, hogy az utasitásokat az %AND kulcsszóval kapcsoljuk össze egyetlen utasitássá. Záró kulcsszó szerepel a GESAL és a XXPL nyelvekben (DO ... OD, ill. DO ... END).

b/ Az F1 feltétel vizsgálata a

```
WHILE   F1   DO   U1;
```

utasitás esetén a ciklustörzs lefutása előtt történik, mig a

DO U1 WHILE F1;

utasitásnál a ciklustörzs lefutása után.

A PLM/R10 nyelvben lehetőség van arra, hogy a feltétel vizsgálata a ciklustörzs végrehajtása közben történhet. Formája:

(WHILE U1 WHILE F U2 WHILE);

itt U1 és U2 utasitássorozat, F feltétel. F nem teljesülésekor csak U1 hajtódik végre és a program a WHILE utáni utasitáson folytatódik.

c/ A ciklus befejezését WHILE kulcsszó esetén a feltétel hamissá válása, UNTIL kulcsszó esetén pedig a feltétel igazzá válása okozza.

Ezek a lehetőségek az összes lehetséges kombinációban előfordulnak az egyes SIL nyelvekben. Mégpedig:

WHILE ... DO ...	minden SIL nyelvben
UNTIL ... DO ...	BCPL, BLISS
DO ... WHILE ...	BCPL, BLISS, C-nyelv, PLM, IMP
DO ... UNTIL ...	BCPL, BLISS, GESAL, PASCAL

Természetesen a szintaxis eltérő az egyes SIL nyelvekben, de a szemantika igy foglalható össze.

Ciklusutasitás ciklusváltozóval és szabályos ismétléssel (FOR)

Hagyományos ALGOL-60 beli alakja:

FOR N = K1 STEP K2 UNTIL K3 DO U;

itt N a ciklusváltozó neve, K1 a kezdőértéket, K2 a lépésközt, K3 a végértéket megadó kifejezések, K pedig utasitás(sorozat), a ciklus törzse.

Ennek a szerkezetnek szintén különböző variánsai vannak a különböző SIL nyelvekben.

Az U utasitás lehet egyetlen összetett utasitás a BEGIN ... END vagy IMP-nél a %CYCLE ... %REPEAT kulcsszavak között, de zárhatja az utasitást záró kulcsszó, pl. OD a GESAL-ban, vagy END az XXPL esetében.

K1, K2 és K3 általában kifejezések, pl. GESAL, C-nyelv, XXPL, IMP nyelvekben. A kifejezések kiértékelése a ciklusba való belépés előtt történik meg. Van olyan nyelv, ahol nem mindenütt állhat kifejezés (PL360, BCPL, PLM), hanem csak konstans.

A lépésköz megadása néhány nyelvben elhagyható (XXPL, BCPL), ez esetben K2 default értéke = 1. A PASCAL nyelvben nincs lehetőség lépésköz megadására, itt a lépésköz mindig = 1. A GESAL nyelvben K1, K2 és K3 is elhagyható, default értékük rendre 0, 1 és ∞.

Néhány nyelvben (XXPL,GESAL,PASCAL, BLISS,PLM/R10) a ciklusváltozó mozgási irányát is jelezni lehet (TO, DOWNTO; INCR, DECR). Ez tulajdonképpen redundáns eszköz, mivel a lépésköz negativ értéket is felvehet, ez azonban csak futáskor derül ki. A DOWNTO kulcsszó bevezetése optimálisabb forditást tesz lehetővé, másrészt a program olvashatóságát is javitja.

Általános ciklusutasitás

A FOR és a WHILE utasitások a GESAL nyelvben összeolvadnak. Az általános ciklusutasitás itt:

FOR N1 FROM K1 TO K2 BY K3
WHILE F1 DO U1 UNTIL F2 OD.

A nevek jelentése mint az előzőekben.

Az utasitás bármely része elmaradhat és igy a közönséges FOR és WHILE utasitásokat kapjuk.

3.3.3 Esetszátválasztás

A modern nyelvekben szereplő esetszétválasztó utasitás a CASE. A hagyományos nyelvekben a több irányu elágazást más eszközökkel oldják meg. Erre szolgál pl. az ALGOL-60 kapcsoló (switch) fogalma, a FORTRAN nyelv kiszámitott GOTO utasitása vagy a BSIC nyelv ON utasitása. Az utóbbi kettő szemantikája szerint a CASE utasitás elődjének tekinthető. Mivel a SIL nyelvekben vagy a kapcsoló vagy a CASE utasitás szerepel, ezért mi ezzel a kettővel foglalkozunk.

A kapcsoló (switch)

A kapcsoló fogalma legtisztábban az IMP nyelvben szerepel. A kapcsoló a nyelv egy objektuma, amelyet a változókhoz hasonlóan deklarálni kell. A deklaráció megadja a kapcsoló nevét és egy egész intervallumot, amelybe a kapcsoló értékének bele kell esnie.

Pl. %SWITCH NAME (1:10)
Az utasitások pedig NAME(I): alaku cimkét viselhetnek, ahol I a switch intervallumába eső egészkonstans.
Pl. NAME(5):
Végül a vezérlésátadás
-→ NAME (int. expr.)
alaku, ahol int. expr. értékének bele kell esnie a switch intervallumába. Nem szükséges, hogy az intervallum minden elemehez legyen cimke rendelve, de ha nem létező cimkére akarunk ugratni, akkor hibajelzés történik.

Az esetszétválasztó utasitás (CASE)

Általános alakja:

```
CASE          K1   OF
  C1    :     U1
  :
  Cn    :     Un
DEFAULT :     UN +1
  END
```

Itt K1 a szelektor kifejezés; C1, ..., Cn cimkék szelektor tipusu konstansok; U1, ..., Un+1 végrehajtható utasitások. Végrehajtáskor a szelektor kifejezés kiértékelődik és az az utasitás kerül végrehajtásra, amelynek cimkéje megegyezik a szelektor értékével. Ha nincs ilyen cimke, akkor a DEFAULT cimkével ellátott utasitás kerül végrehajtásra. Az END záróutasitás az esetek felsorolását zárja le. Az aktuális utasitás végrehajtása után az END után következő utasitás kerül végrehajtásra.

Az utasitás változatait az adja, hogy milyen lehet a szelektor kifejezés tipusa, hogy az esetek szétválasztása cimkékkel történik-e, hogy az esetek leirása egyetlen összetett utasitás vagy egy utasitássorozat, hogy van-e default cimke és hogy az esetfelsorolást mi zárja le.

A szelektor kifejezés tipusa értelemszerüen int, boolean, character, enumeráció és subrange lehet, ha van ilyen tipus a nyelvben (PASCAL, GESAL). Több nyelvben nincs tipus megszoritás.

Az esetek szétválasztása a legtöbb nyelvben cimkékkel történik (GESAL, BCPL, PASCAL). Ha nincs cimke, akkor az esetek felsorolási sorrendje a döntő. Ez esetben viszont a szelektortipus értelmezési tartományának minden eleméhez esetet kell rendelni. Pl. XXPL, PL360, PLM/R10.

Az esetek leirása egyetlen összetett utasitás azokban a nyelvekben, ahol van összetett utasitás (PASCAL, XXPL). Másutt utasitássorozat valami záró kulcsszóval lezárva (pl. PLM/RIO-ben a CASE), GESAL nyelvben az OF ismétlődése). Ismét másutt az esetek tartalmazhatják egymást, csak a belépési pont más (BCPL, C-nyelv).

Default cimke több nyelvben szerepel (GESAL, BCPL, C-nyelv), más nyelvekben nincs ilyen (PASCAL, XXPL). Az utóbbi esetben, ha a szelektor értéke nem esik a tervezett tartományba, akkor az utasitás hatása definiálatlan, pl. az XXPL nyelvben ellenőrizhetetlen ugrás következik be. Tehát default cimke bevezetése nagyon célszerü.

Az esetek felsorolását záró kulcsszó zárja (END, ESAC), hacsak nem az összes eset egyetlen összetett utasitást alkot (BCPL).

3.3.4 Feltétel nélküli vezérlésátadás (GOTO)

A strukturált programozás hivei nagyon károsnak tartják a feltétel nélküli vezérlésátadást. Ennek ellenére a BLISS nyelvet kivéve mindenütt megtarották. Néhol azzal nehezitik a használatát, hogy a cimkéket explicit deklarálni kell, pl. PASCAL, MARY. A feltétel nélküli vezérlésátadás alakja:

GOTO cimke

vagy GOTO expr. . (GESAL, C-nyelv, BCPL).

Kifejezés csak azokban a nyelvbekben irható a GOTO utasitás argumentumába, ahol cimkeváltozó szerepel vagy másfajta eszköz van a cimek kezelésére.

3.3.5 Szökéskifejezések

A programozási gyakorlatban gyakran fordul elő olyan szituáció, hogy egymásba ágyazott ciklusok esetén egy belső ciklusból a beágyazó ciklusok végigjárása nélkül akarunk kiugrani. Ilyen esetekben a strukturált vezérlésátadási szerkezetek mellett a GOTO utasitást használjuk. Azokban a nyelvekben, ahol GOTO utasitás nincs, az un. "szökéskifejezések" segitségével valósithatjuk meg ezt. Legjobb példa erre a BLISS nyelv, amelyben a következő szökéskifejezések vannak:

```
exitblock E
exitcompound E
exitloop E
exitset E
exitcase E
exitselect E
exit E
```

A szökéskifejezések hatása a megnevezett vezérlési környezetből (blokk, zárt kifejezéssorozat, ciklus) való kilépés. Exit hatására bármelyik vezérlési környezetből kilép. E lesz a kilépsi érték.

Hasonló szökésutasitásokat találunk más nyelvekben, pl. a BCPL nyelvben a FINISH, BREAK és bizonyos értelemben az ENDCASE, valamint a C-nyelvben a BREAK és a CONTINUE. Mindkettőnél a BREAK az őt tartalmazó legszükebb ciklusból való kilépést okozza. CONTINUE a ciklus ismétlődési pontjára adja a vezérlést.

3.3.6 Rutinhivás és visszatérés

A rutin (eljárás vagy függvény) hivás hatására megtörténik a paraméteradás és a vezérlés a rutin törzsének első végrehajtható utasitására adódik.
Az eljárás hivás általában

```
CALL név (P1, P2, ..., Pn)
```

vagy `név (P1, P2, ..., Pn)`
alaku és önálló utasitás, mig a függvényhivás mindig

név (P1, P2, ..., Pn)

alaku és kifejezés elemeként szerepel.

P1, P2, ..., Pn az aktuális paraméterek.

A paraméter-átadás formáit és a rekurzió fogalmát 3.1.4-ben már tárgyaltuk.

Legáltalánosabban paraméter lehet:

konstans,

egyszerü változó,

tömbre vagy rekordra mutató pointer,

eljárás vagy függvény.

Pl.: PASCAL, IMP.

Szükebb kiépitésben függvényparaméter nincs, csak proc tipusu változó (pl. GESAL) vagy függvényre mutató pointer (C-nyelv). Ha a nyelv csak adatra mutató pointer tipust enged meg, vagy azt sem, akkor ez a lehetőség elesik és csak változó és érték paraméter van, Pl. XXPL.

A rutinból való visszatérést vagy utasitás (RETURN, RESULT stb.) valósitja meg, vagy a rutin fizikai vége jelenti a kiléPési pontot is. A visszatérő érték lehet a RETURN utasitás argumentuma (pl. XXPL), vagy előre kijelölt változó (pl. GESAL).

3.3.7 Korutin hivás

A korutin fogalom a SIMULA'67 nyelvvel kapcsolatban vált közismertté hazánkban. Az általunk vizsgált nyelvek közül egyedül a BLISS-ben szerepel korutin.

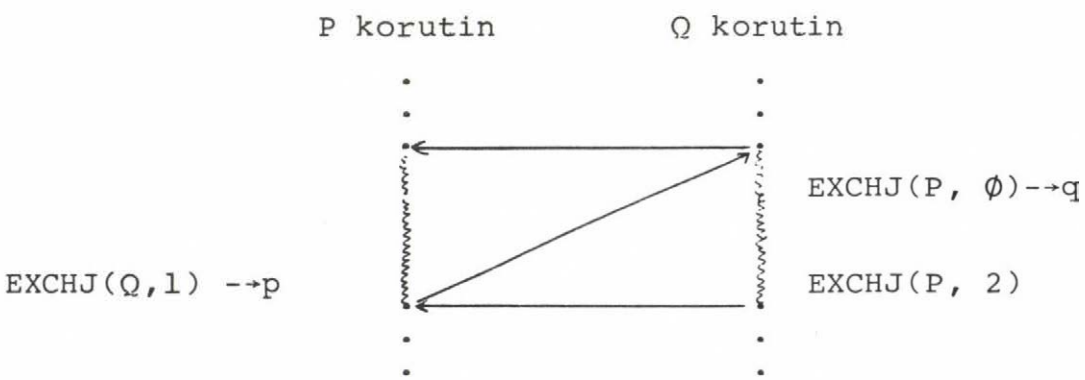
A korutinok kvázipárhuzamosan futó rutinok, amelyek hivhatják egymást és adatot adhatnak át egymásnak. Korutin hiváskor a pillanatnyi végrehajtási hely megőrződik és ha ismét visszakapja a rutin a vezérlést, akkor onnan folytatódik a végrehajtás.

A BLISS-ben korutin mechanizmus a következőképpen müködik: Korutin létrehozására a CREATE kifejezés szolgál. Bármelyik függvény törzse korutinként is aktiválható. Ugyanannak a törzsnek tetszőleges számu példánya létrehozható. A CREATE kifejezés hatására létrejön egy uj példány, terület foglalódik a számára, de a függvény törzse nem fut le. A függvénytörzs kezde-

te lesz az aktiválási pont és a következő kifejezés kiértékelése következik. A már létrehozott korutinra korutinhivással adható át a vezérlés.

Korutinhivásra az EXCHJ (E5, E6) kifejezés szolgál, amelynek hatására a vezérlés az E5 által megnevezett korutinra adódik és E6 pedig az átadandó érték, ez annak az EXCHJ kifejezésnek az értéke lesz, amelynek hatására az E5 által megnevezett korutinból legutoljára kiléptünk.

Pl.



Ebben az esetben q = 1 és p = 2 lesz.

4. A SIL NYELVEK ISMERTETÉSE

A SIL nyelvek ismertetését a következő 4 csoportra bontva végezzük:

- alacsony szintü gépközeli nyelvek:
  PL360, PLM/R10
- középszintü tipus nélküli nyelvek:
  BCPL, BLISS
- középszintü tipusos nyelvek:
  XXPL, IMP, C-nyelv, GESAL, PASCAL
- magasszintü nyelvek:
  MARY, MODULA, TARTAN

4.1 Alacsony szintü gépközeli SIL nyelvek

Erre a nyelvcsoportra az jellemző, hogy a nyelv gépközeli, azaz lehetővé teszi a computer minden hardware lehetőségének kihasználását. A nyelv gépfüggő, használatához szükséges a gép assembly nyelvének és hardware adottságainak, pl. az IT rendszer, ismerete. Az utasitások igazodnak a gépi utasitáskészlet szemléletéhez. Gépi utasitások is beépithetők a programba. Ebből következik a nagyfoku átláthatóság. A programozó tudja, hogy a leirt utasitás milyen assembly utasitásokká fordul. A fenti sajátságok miatt lehetővé válik hatékony tárgykód generálása. Viszont ugyancsak a fentiekből következik, hogy ezek a nyelvek nem portabilisek. A programozás alacsony szintü SIL nyelvekben sokkal kényelmesebb, mint assembly nyelven és a program is olvashatóbb, áttekinthetőbb.

4.1.1 PL360

A PL360 az IBM 360 gépcsalád alacsony szintü rendszerprogramozási nyelve. A nyelvet N. Wirth specifikálta 1965-ben, azóta széles körben elterjedt az IBM felhasználói között, pl. az IBM/SIMULA-t is ezen a nyelven irták.

A nyelv igazodik az IBM 360 hardware lehetőségeihez, utasitáskészletéhez és assembly szintü nyelvének szemléletéhez. Fontos szerepük van a regiszterekneк. 16 db 32 bites és 4 db 64 bites regiszter áll a programozó rendelkezésére. Ezek gyorsak; aritmetikai, bitenkénti logikai és shift müveletek végezhetők rajtuk és bázisregiszterként használhatók a cimzési mechanizmusban. Az utasitásokban bázisregiszterhez relativ cimek szerepelnek, bázisregiszterként pedig bármelyik 32 bites regiszter kijelölhető. Egy bázisregiszterrel csak 4095 byte relativ környezet cimezhető.

A PL360 objektumai regiszterek, cellák, függvények és procedure-k. A regiszterek a gépi regiszterekнek megfelelő tárolók, integer (32 bit), real (32 bit) és longreal (64 bit) tipusu regiszter van. A cellák a memóriarekeszeknek megfelelő tárolók. Öt tipusba sorolhatók: byte (8 bit), shortinteger (16 bit), integer (32 bit), real (32 bit) és longreal (64 bit). A függvények alatt itt nem a magas szintü nyelvekből ismerős függvényfogalmat értjük. Itt egy függvény tulajdonképpen egyetlen gépi kódu utasitást reprezentál. A procedure egy utasitássorozat, a hagyományos szubrutin fogalomnak nagyon erősen korlátozott változata, formális paraméterei nincsnek és semmiféle deklaráció nem lehetséges benne. Rekurzió megengedett ugyan de mivel lokális változók nincsenek, ezért stack technika nem szükséges az implementációhoz, egyedül a visszatérési cimeket kell számon tartani.

4.1.1.1 A PL360 adatszerkezete

Az adatok konstansok, változók és tömbök lehetnek.

A konstansok lehetnek byte vagy character, short integer, integer, real és longreal tipusuak és mindegyik lehet decimális vagy hexadecimális felirásban.

A változók lehetnek regiszterek vagy cellák. A regiszterekнek standard azonositóik vannak, ezeken hivatkozhatók, mégpedig:

R0, R1, ..., R15	integer regiszterek
F0, F2, Fe, F6	real regiszterek
F01, F23, F45, F67	longreal regiszterek.

A cellákat deklarálni kell.

Deklaráció: <tipus> <azonositó lista>

<tipus>::= BYTE|SHORT INTEGER|INTEGER|REAL|LONG REAL

Pl. BYTE FLAG

LONG REAL A, B, C

A tömbök csak egydimenziósak lehetnek és csak cellákból épülhetnek fel, regiszterekből nem.

Deklaráció: ARRAY M T N1, N2, ...

Itt M a méret, csak nem negativ egészszám lehet, T tipus és N1, N2, ... cellanevek. A tömbelemek számozása 0-tól kezdődik és byte-onként történik még akkor is, ha a tömb elemei nem BYTE tipusuak. A tömb elemeihez való hozzáférés indexelt cellanévvel történik.

Pl. az ARRAY 3 INTEGER T1

tömb elemeihez való hozzáférés a T1(0), T1(4), T1(8) hivatkozással történik. Az index egész konstans, egész regiszter vagy egész regiszter + egész konstans lehet.

Kezdőérték cellának vagy tömbnek adható. Cellának konstans, string vagy cim kezdőérték adható, tömbnek pedig ilyen elemekből felépülő lista. A listán belül zárójelezés és ismétlési tényező megengedett. Lehetőség van implicit tömbméret megadására. Ilyenkor a tömbméret helyett a DATADEF kulcsszót kell használni.

A regisztereken és cellákon végzett müveletek megfelelnek a load, store, valamint az aritmetikai, bitenkénti logikai és shift assembly utasitásoknak. A cella-értékadás (store müvelet) formája: <cella-hivatkozás>:= <regiszternév>. Tipus egyezés szükséges, de integer short integerbe és longreal realbe is beirható. Az egyszerü regiszter értékadás (load müvelet) formája: <regiszter>:= konstans>|<regiszter>|<cella-hiv>|@ <cella-hiv>. @ cella-hivatkozás esetén a cella abszolut cime kerül a regiszterbe (LOAD ADDRESS utasitás), ily módon burkoltan pointer változónk van. Tipus egyezés szükséges, de short integer beirható integerbe és real pedig longrealbe. Az általános regiszterértékadás ugy jön létre, ha regiszter értékadás jobb oldalához egy operátort, majd egy operandust irunk és ezt akármeddig foly-

tathatjuk. A nyelvben aritmetikai, logikai és shift operátorok szerpelnek. Az operandus konstans, regiszter vagy cellahivatkozás lehet. Az operátorok tipusfüggőek, mégpedig:

Operandus / Operátor	opl és eredmény	op2
aritmetikai	regiszter-értékadás	konstans, regiszter, cella-hivatkozás
logikai	egész reg. értékadás	egész konstans, egész reg., egész cella-hiv
shift	egész reg. értékadás	egész konstans, egész regiszter

A tipusegyeztetés ugyanugy történik, mint az egyszerü regiszter értékadasnal.

Az aritmetikai operátorok közül kiemeljük a =: operátort (átutalás), amely a bal oldalán álló regiszter operandus tartalmát átviszi a jobb oldalon lévő regiszter vagy cella-hivatkozásba, ezáltal egy kifejezés kiszámitása közben módunk van arra, hogy a részeredményeket lerakjuk valahová későbbi felhasználás céljából.

A regiszter értékadások végrehajtása balról jobbra haladva történik, a részeredmények mindig a kiindulási regiszterben vannak.

Példa: R3:= i + 1 =: i + 6 értékadás lépései:

R3 - i, R3 - R3 +1, R3 ->i, R3<- R3 +6.

Az általános regiszter értékadás felel meg más nyelvek airtmetikai kifejezésének.

A cella- és regiszter-értékadás mechanizmusa mutatja, hogy a nyelv mennyire gépközeli és gépfüggő, de egyben azt is mutatja, hogy hatékony kód generálását teszi lehetővé.

4.1.1.2 A PL360 vezérlési szerkezetei

Feltételes vezérlésátadás

IF F1 THEN U1

és IF F1 THEN U2 ELSE U3

Itt U1 és U3 tetszőleges utasitás (blokk), U2 viszont nem lehet FOR, IF és WHILE utasitás.

Az F1 feltétel a következő lehet:

- feltételkód vizsgálat (hardware),
- reláció (a bal oldalon csak regiszter állhat),
- byte-test (a byte tartalma =#FF?),
- összetett feltétel (a fenti feltételek AND vagy OR operátorokkal összekapcsolva, az AND és OR vegyitése tilos).

Ciklus utasitások

WHILE F1 DO U1

U1 egyetlen utasitás (blokk), F1 feltétel olyan mint az IF utasitásnál, feltételvizsgálat U1 lefutása előtt. A WHILE utasitásnak más formája nincs.

A FOR utasitás formája:

FOR R: = K1 STEP K2 UNTIL K3 DO U

Itt R: = K1 egészregiszter-értékadás, ciklusvált.: = kezdőérték,
K2 lépésköz = egész konstans,
K3 végérték = regiszter vagy cella-hiv. vagy egész konstans,
U utasitás (blokk).

Esetszétválasztás

Formája:

CASE Ri OF BEGIN U1; ...; Un END

Az Ri egész regiszter tartalma határozza meg, hogy az U1, ..., Un utasitások közül hányadikat kell végrehajtani.

Feltétel nélküli vezérlésátadás

GOTO cimke

Egy utasitásnak több cimkéje is lehet.

Procedure hivás: a procedure nevének leirása.

Paraméterek nincsenek.

4.1.1.3 Programszerkezet

A PL360 blokk utasitása a hagyományos blokk és az összetett utasitás ötvözete. Formája:

BEGIN D1; D2; ...; Dk; U1; U2; ...; Un; END

ahol Di deklaráció, Ui pedig utasitás.

A deklarációk elmaradhatnak, de legalább egy utasitásnak kell lennie a blokkban. A PL360 blokk nyitott scope a regiszter, cella, függvény és procedure deklarációk és a cimkék szempontjából is.

A szegmens fogalma: PL360-ban szegmensnek nevezünk egy olyan program vagy adatrészletet, amelynek cimzéséhez ugyanazt a bázisregiszter-értéket használjuk. Mivel egy bázisregiszter segitségével csak 4095 byte relativ környezet cimezhető, ezért hosszabb programok és adatok esetén szegmentálni kell. A szegmensek formálisan eljárások, amelyeket meg lehet hivni, cimzésük teljesen független egymástól. Ez lehetővé teszi a program részenkénti forditását és különböző nyelveken irt programok (pl. FORTRAN) PL360 programmal való összeszerkesztését is. A szegmentálás tehát egyrészt kényszerüség, másrészt jelentős előnyökhöz juttat.

A szegmens formája:

SEGMENT PROCEDURE N Ri U

ahol N = a szegmens neve, Ri = a bázisregiszter, U = utasitás.

A részenkénti forditást elősegitő eszköz a GLOBAL, EXTERNAL és ENTRY procedure bevezetése. A GLOBAL procedure leforditódik, de nem hivható, felhasználása egy másik PL360 programból történhet, ha ott EXTERNAL-ként deklaráltuk. EXTERNAL procedure deklaráció esetén tárgykód nem generálódik. Az ENTRY procedure ugyanabból a programból és külső szegmensből is hivható.

4.1.1.4 Egyéb sajátságok

Függvények. A nyelv a gépi kódu utasitásokat függvényeknek tekinti, amelyek a regiszterek és cellák tartalmától függően valamilyen uj tartalmat rendelnek hozzájuk. A függvényeket részben definiálni lehet, részben standard függvényhivás név (paraméterlista) alaku és egyetlen gépi kódu utasitássá fordul.

Szinonim deklaráció. A szinonim deklaráció arra szolgál, hogy egy regiszterhez vagy cellához különböző uton is hozzáférhessünk.

A szinonim regiszter deklaráció formája:

T REGISZTER N1 SYN Ri, N2 SYN Rj, ...

Itt T tipus, N1, N2 nevek, Ri, Rj regiszterek.

Példák: INTEGER REGISTER Ni SYN R1, Nj SYN R2.

REAL REGISTER REAL1 SYN F2.

A szinonim cella-deklarációnak különböző tipusai vannak:

T N SYN K — T = tipus, N = név, K = egész konstans.

Pl. INTEGER I SYN # 50 - abszolut cim megadása

T N SYN CH — T = tipus, N = név, CH = cella-hiv.

Pl. INTEGER J SYN H(4) - itt H INTEGER tömbnek volt deklarálva.

4.1.2 PLM/R10

A PLM/R10 az R10 számitógép alacsony szintü rendszerprogramozási nyelve, amelyet Mandler György fejlesztett ki 1975-ben az MTA SZTAKI-ban, ahol azóta is használják. A nyelv a magas szintü nyelvekből átvett programozási eszközök és az assembly szintü lehetőségek összeötvözése. A forditóprogram tulajdonképpen egy makrokészlet, amelynek felhasználásával az MP/3 makroprocesszor a forrásnyelvi programot jó hatásfoku assembly nyelvü programmá alakitja át. Hatékony használatához szükséges az IDOS, azaz a müködtető operációs rendszer, az MP/3 makroprocesszor, az R10 assembler és a gép architekturájának alapfoku ismerete.

Az R10 hardware-jének legfontosabb sajátságai, hogy három aritmetikai regiszter, A, E, X áll a programozó rendelkezésére, ezek mindegyike más funkciót tölt be és más müveletek végezhetők rajtuk. A müveletek lehetnek byte-osak, szavasak vagy duplaszavasak. Az A regiszterben a rövid (byte és szó), az A, E-ben a hosszu (duplaszavas) aritmetikai müveletek hajtódnak végre, mig X az indexregiszter. A cimzési mód bázis-relativ. Két bázisregiszter van, globális és lokális.

Az R10 assembly nyelvü program szekciókból áll, mégpedig egy közös adatszekcióból (CDS), névvel ellátott lokális adatszekcióból (LDS) és ugyancsak névvel ellátott programszekciókból (LPS). Egy LPS az általa megnevezett LDS-re és a közös CDS-

re hivatkozhat. Egy LDS-re több LPS is hivatkozhat. Egy adatszekcióból csak 256 byte direkt elérésü, a többire csak indirekt lehet hivatkozni.

4.1.2.1 A PLM/R1O adatszerkezete

Az A, E, X regiszter standard azonositója (A), (E) és (X). Az adatok leirására szolgáló direktivák hasonlóan az assembly szintü adatleiró direktivákhoz helyfoglalást, esetleg kezdőértékadást végeznek. A konstansok tipusa lehet byte, szavas egész, duplaszavas egész, lebegőpontos szám és string. A változókra vonatkozóan csak helyfoglalás történik a megadott méret szerint, a változóknak nincs tipusa és tipus ellenőrzés sincs. Valamennyi adatdeklaráló direktiva után az adatok vesszővel elválasztva, egymás után irhatók a sor végéig.

A következő adatdeklaráló direktivák vannak:

DATA direktiva direkt elérésü szavas egészek, duplaszavas egészek vagy lebegőpontos konstansok elhelyezésére és indirekt elérésü tömbök helyfoglalására.

BYTES direktiva byte-os adatok elhelyezésére.

EQUALS direktiva névvel ellátott konstansok (literál) deklarálására.

Tömbdeklarálás akár DATA, akár BYTES direktivával történhet oly módon, hogy a változónév mellé méretet adunk meg vagy egynél több értéket sorolunk fel kezdőértékként.

Pl.: DATA NEV1[6] - NEV1 nevü 6 szóból álló tömb

BYTES NEV2 ="SZOVEG" - NEV2 nevü 6 byte-ból álló tömb

A tömbhivatkozás indirekt indexelt cimzési módu assembly utasitássá fordul.

A NÉV(INDEX) kifejezésben a NEV a tömb cimét, az INDEX pedig az indexregiszter beállitásának módját adja meg. Az utóbbira példák: az (X) pillanatnyi értékével történik az indexelés, (X) értéke növekszik vagy csökken az adott értékkel, (X) értékét egy kifejezés határozza meg stb. Bármilyen változóra kiadhatunk indirekt indexelt cimzési módu utasitást, ami azt jelenti, hogy bármelyik változót tömbnek tekinthetjük.

Értékadás, müveletek

Mivel az assembly szintü utasitások az (A), (E),(X) töltését, tárolását és rajtuk végzett müveleteket tesznek lehetővé, ezért a PLM/R10-ben is ezek a lehetőségek vannak értékadásra.

Az értékadás általános formája:

NA, NE, NX = KA, KE, KX

ahol NA, NE, NX azon változók neve, ahová rendre az (A), (E), (X) értéke kerül.

KA, KE és KX pedig rendre az (A), (E) és (X) értékét meghatározó kifejezések, amelyek a megfelelő regiszterekben végezhető müveleteket tartalmazhatnak.

A fentivel teljesen equivalens forma:

KA, KE, KX; NA, NE, NX

Az általános képlet tetszőleges része elhagyható, igy az értékadásnak speciális formáit kapjuk meg. A képletek azonositásához szükséges elválasztó jeleket azonban nem szabad elhagyni.

A változó értékadás az általános képlet megfelelő része:

NA, NE, NX = vagy ;NA, NE, NX

Lehetőség van egy értéknek több változóba való betöltésére is.

Példák:

A1 = vagy ; A1 tárold (A)-t A1-be

;,, I1; , , I2 tárold (X)-t I1-be és I2-be.

A regiszter értékadás formája:

= KA, KE, KX vagy KA, KE, KX;

Ha csak (A) értékét állitjuk, akkor a többi rész elhagyható, az (E) és (X) állitásánál azonban ki kell tenni a vesszőket.

Példák: =KA kerüljön KA (A)-ba

=,,KX kerüljön KX (X)-be.

A KA, KE és KX kifejezésekben aritmetikai müveletek, gépi utasitások kijelölése és az operandus értelmezését befolyásoló jelölések szerepelnek. Mint már emlitettük, különböző regiszterekre különböző müveletek vonatkoznak. Néhány müveleti jelnek más az értelmezése aszerint, hogy (A)-ra vagy (X)-re vonatkozik.

Ezért a kifejezéseknél meg kell különböztetni, hogy mely regiszterre vonatkoznak.

Gépi utasitások jelölhetők ki KA-ban és KE-ben pontok közé irt hárombetüs mnemonikkal és az utána következő operandussal.

Pl. =A1.>LS.8 A1 jobbra lép 8 bittel logikailag.

Az operandus értelmezését befolyásoló jelölések egyrészt az operandus tipusára vonatkoznak (byte, duplaszó, cim stb.), másrészt az indirekt indexelt hivatkozásokat teszik lehetővé. Direkt indexelt hivatkozás nincs a hardware-ban.

4.1.2.2 A PLM/R10 vezérlési szerkezetei

Minden vezérlési szerkezetre jellemző a zártság, amelyet kerek zárójelpár biztosit és a kezdő kulcsszó megismétlődése a végzárójel előtt.

Pl.: (IF ... IF).

A feltétel lehet egyszerü és összetett. Az egyszerü feltétel alakja:

K R V

ahol K egy (A)-ra vonatkozó kifejezés,

R reláció a <, >, = jelek és kombinációik,

V viszonyitási alap: változó, konstans és hardware indikátor (carry, overflow).

A feltétel egyes részei elmaradhatnak. Ha K elmarad, a feltétel (A)-ra vonatkozik. Ha V marad el, akkor a feltétel az indikátorokra vonatkozik. Az elemi feltételek az AND, OR és NOT logikai operátorokkal összefüzhetők. A kiértékelés sorrendjét zárójelezéssel előirhatjuk.

Az IF utasitások formája:

(IF F) (P)

vagy (IF F1 P1 ELSE P2 IF)

Itt F, F1 feltételek, P, P1 és P2 tetszőleges programrészek.

Több feltétel soros vizsgálatára a következő forma szolgál:

(IF F1 P1 IF F2 P2 ... Fn PN ELSE PE IF)

Az Fi feltétel teljesülése esetén a Pi programrész kerül végrehajtásra, majd az IF) utáni soron folytatódik a program.

Ha az F1, F2, ..., FN feltételek egyike sem teljesül, akkor PE kerül végrehajtásra.

Az IF strukturák egymásba ágyazása a következőképpen történhet:

((IF F1 P1 (IF F2 P2 ... (IF FN PN ELSE PNE IF))

Ha az F1, F2, ..., F(N-1) feltételek valamelyike nem teljesül, akkor az IF))-et követő programrész kerül végrehajtásra. Ha F1, F2, ..., F(N-1) teljesülnek, akkor FN-től függően PN vagy PNE hajtódik végre, majd az IF))-et követő program következik.

Ciklus utasitások (WHILE, FOR).

A WHILE utasitásnál a feltétel vizsgálata a törzs lefutása előtt, után és közben is történhet.

Formái:	feltételvizsgálat
(WHILE F P WHILE)	elöl
(WHILE P WHILE) F	hátul
(WHILE P1 WHILE F P2 WHILE)	középen

Itt F feltétel, P, P1 és P2 programrészek. Ismétlés a feltétel teljesülésekor történik.

A FOR utasitás formája:

(FOR N = K1 STEP K2 UNTIL K3 P FOR)

Itt N változónév K1 kifejezés, K2 és K3 pedig változók vagy konstansok, P program. Ha K2 helyett -K2 szerepel, akkor a ciklusváltozó dekrementálása történik.

Az XCYCLE utasitás egy ciklus végrehajtása (X)- nek mint ciklusváltozónak szabályos csökkentésével. A kilépés feltétele (X) értékének nullán való áthaladása. Formája:

(XCYCLE K1 P XCYCLE STEP K2)

Itt K1 csak (X)-re vonatkozó kifejezés lehet, K2 változó vagy konstans, P program. (X) értékének csökkentése és a kilépés vizsgálata P végrehajtása előtt történik.

STEP K2 elmaradhat, default értéke K2 =1.

Az esetszétválasztó utasitás alakja:

```
(CASE K1 OF K2
 CASE P∅
 :
 CASE P(K2-1)
 CASE)
```

Itt K1 (A)-ra vonatkozó kifejezés, K2 az esetek száma, konstans. PØ, ..., P(K2-1) az eseteknek megfelelő programrészek. Cimkék nincsenek, az esetek szétválasztását a CASE kulcsszó ismétlése végzi és sorrendjük a döntő. K1 < 0 vagy K1 > K2-1 esetén az utolsó eset hajtódik végre, amely igy vészkijáratként is szolgál.

A feltétel nélküli vezérlésátadásra a GOTO utasitás szolgál. Argumentuma csak cimke lehet. A program tetszőleges sora cimkézhető a kivánt sor elé, külön sorba irt LABEL direktivával.

Példa:
```
LABEL CIM1
  U1
  :
  GOTO CIM1
```

Szubrutin hivás. Egy szekción belül létesitendő szubrutin paraméterátadásra a regisztereken keresztül történik.

A szubrutin deklarációja:

```
SUBROUTINE N1 P1 END OF N1
```

Itt N1 név, P1 program.

Hivása:

```
GOSUB N1 IA, IE, IX; OA, OE, OX
```

Itt IA, IE, IX input paraméterek a regiszteréknek megfelelő kifejezések, OA, OE, OX pedig az output regiszterértékek tárolási helyei.

Szekcióhivás és visszatérés

A hivás formája:

```
CALL NÉV IA, IE, IX; OA, OE, OX
```

Paraméterátadás mint a szubrutinnál.

Visszatérés programszekcióból:

```
RETURN OA, OE, OX
```

Itt OA, OE és OX kifejezések a visszatérési értékek.

Supervisor hivás:

```
CSV NÉV IA, IE, IX; OA, OE, OX
```

Paraméterátadás mint a szubrutinnál.

A STOP utasitás a program futásának befejezését irja elő.

4.1.2.3 A PLM/RlO program programszerkezete

A PLM/RlO nyelvü program szerkezete az assembly nyelvü programok szerkezetéhez hasonló. A következő programegységek vannak: A program szekciók, a szubrutinok és a supervisor szekciók. A szubrutinokról már volt szó. A supervisor szekciók az operációs rendszer olyan részei, amelyeket a program hivhat. A programszekciók közül a közös adatszekciót nem jelöli külön direktiva, ezzel kezdődik a program. A lokális szekciókat LOCALS direktiva vezeti be, amelynek argumentuma a lokális adatszekció neve. A programszekciókat pedig

PROGRAM Pl LOCALS Nl

alaku direktiva vezeti be, ahol Pl a programszekció neve, Nl pedig egy lokális adatszekció neve. Pl a közös adatokon és Nl adatain dolgozhat. A programszekciók a már ismertetett CALL utasitással hivhatják egymást.

Részenkénti fordithatóság a PLM/RlO-ben nincs.

4.2 Középszintü tipus nélküli nyelvek

Ezekről a nyelvekről 3.2.3-ban már beszéltünk, főleg az adatszerkezettel kapcsolatban. Itt ezen nyelvek egyéb tulajdonságait ismertetjük. Közös jellemzőjük, hogy nagy szabadságot adnak a programozónak, ami viszont nagy veszélyeket is rejthet magába.

4.2.1 BLISS

A nyelvet W.A.Wulf, D.B. Russell és A.N.Habermann készitette a Carnegie Mellon Egyetemen 1969-70-ben.

A nyelv implementációs nyelvnek készült, software rendszerek irására a PDP-10 számitógéphez. A tervezés fő szempontjai voltak, hogy nagy hatékonyságu tárgykódot lehessen vele készitteni, hogy lehetőség legyen minden jelentős hardware sajátság kihasználására és hogy a programok módosithatósága, fejleszthetősége biztositott legyen. A szerzők szerint a fő sajátság, ami elősegiti ezen célok megvalósitását az a mechanizmus,

ami lehetővé teszi az adatstrukturák reprezentációjának definiálását a struktura elemeihez való hozzáférési algoritmus megadásával. Programszerkezete ALGOL-60 szerü, blokkstrukturával és rekurziv eljárásokkal. Vezérlési szerkezetei a szokásos feltételes, ciklus és esetszétválasztó szerkezetek mellett korutinokat és szökéskifejezéseket is tartalmaznak. Feltétel nélküli vezérlésátadás nincs a nyelvben. Jellemzője még, hogy kifejezés nyelv.

4.2.1.1 A BLISS adatszerkezete

A BLISS adatszerkezetének két legfontosabb jellemzője, hogy tipus nélküli és hogy lehetőség van az adatszerkezetek elemeinek elérési mechanizmusának definiálására.

A tipus nélküliséget 3.2.3-ban tárgyaltuk. A BLISS adategysége a társzegmens. Egy társzegmens rögzitett és véges számu szóból áll, amelyek mindegyike rögzitett és véges számu bitet tartalmaz. Egy szón belül a bitek folytonos sorozatát mezőnek nevezzük. A mező megnevezhető és a név értéke a megfelelő mező cime. Speciálisan egy teljes szó is mező és igy megnevezhető. A gyakorlatban egy társzegmens vagy programot vagy adatokat tartalmaz. Az adatok egész számok, lebegőpontos számok, karakterek vagy cimek lehetnek. A BLISS nyelv azonban ezeket az adatokat bitmintáknak tekinti. A rajtuk végezhető müveletek a szokásos aritmetikai, bitenkénti logikai és relációmüveletek. A relációk értéke = 1, ha igaz a reláció, különben = 0.

A társzegmensek megnevezése a deklarációban történik, amely a néven kivül a szegmens hosszát is specifikálja (default méret = 1). A deklaráció kulcsszava pedig tulajdonképpen a memória osztályt adja meg, amely a következő lehet: GLOBAL, OWN, LOCAL, REGISTER, FUNCTION. Az OWN memória osztály a 3.2.9-ben használt terminológiával a statikus memória osztálynak felel meg. A FUNCTION deklaráció létrehoz egy szegmenst, amelynek neve a függvény neve, tartalma a megfelelő gépi kódu program, hossza az ehhez szükséges hossz és amely a program teljes futási ideje alatt megőrződik. A deklaráció a névhez a megnevezett társzegmens első szavának cimét köti és a név a programban

is ezt reprezentálja. A név által megnevezett társzegmens tartalmának elérésére a . operátor szolgál, amely tulajdonképpen egy dereferenciálási müvelet. A következő két deklaráció esetén nem történik helyfoglalás. Az EXTERNAL kulcsszóval deklarált változó egy másik modulban GLOBAL-ként kell hogy szerepeljen és ott foglalódik hely a számára. A BIND deklaráció egy kifejezés értékéhez köt nevet a blokkba való belépéskor.
Példák deklarációkra:

```
GLOBAL G;
OWN X, Y[5], Z;
LOCAL P[100];
REGISTER R1,R2[3];
FUNCTION F(A,B)=. A↑.B;
EXTERNAL S;
BIND Y2=Y+2, PA=P+.A;
```

A mezők megnevezése a következő 5 adattal történik: szócim, kezdő bitpozició, bitek száma, egy (index) regiszter neve és egy "indirect address" bit. Ez mutatja, hogy a nyelv mennyire gépfüggő. A szócim, a pozició és a hossz megadás a következőképpen történik: név<pozició,hossz>. Ha a <...> rész elmarad, a default érték <0,36>. A mező megnevezésben a konstansok helyén állhatnak kifejezések is. P1.E0<E1,E2> jelenti azt a mezőcimet, amelynek szócime E0 (modulo 2^{18}), pozíciója és hossza rendre E1 és E2 (modulo 2^{6}).

Az adatszerkezetek elemeinek elérési mechanizmusának definiálása a STRUCTURE deklarációval történik. Az ugyanazon elérési mechanizmussal rendelkező adatstrukturák ugyanabba a struktura osztályba tartoznak. Ezt a MAP deklaráció adja meg.
A deklarációk formája:

```
STRUCTURE N [P1,P2,...]= E
MAP N N1:N2:...
```

Itt N strukturanév, P1, P2,... formális paraméterek, E kifejezés, amely hivatkozik a P1, P2,... paraméterekre, és N1, N2,... társzegmensek nevei.

1. Példa

```
STRUCTURE ARY2[I,J] = [.ARY2+.I*10+.J];
OWN X[100], Y[100], Z[100];
MAP ARY2 X:Y:Z
```

Ebben a példában definiáljuk az ARY2 kétdimenziós tömb elérési mechanizmust, deklarálunk három egyenként 100 méretü szegmenst és ezeket a szegmenseket besoroljuk az ARY2 struktura osztályba. Ezután az X[E1,E2] hivatkozás az ARY2 struktura deklaráció által definiált elérési algoritmusnak a kiszámitását jelenti a következő helyettesitésekkel: ARY2 = X, I = E1, J = E2.

A BLISS adatstrukturáinak tervezésekor az is cél volt, hogy a strukturadefinció és a struktura elemeivel dolgozó algoritmusok elválaszthatók legyenek, ugyhogy bármelyik módositható legyen a másik megváltoztatása nélkül. A fenti példában a tömb mérete be volt épitve az elérési algoritmusba, ezért ennél általánosabb módszer is szükséges.

Az általános struktura deklaráció a következő:

```
STRUCUTRE N [P1,P2,...] = [E1,E2]E3
```

Itt N a struktura név, P1,P2,... formális paraméterek, E1 a struktura méretét meghatározó kifejezés, E2 egy user-definiált dinamikus tárkiosztási függvény nevét meghatározó kifejezés, E3 pedig az elérési algoritmus. ,E2 és [E1,E2] elmaradhat.

2. Példa

```
STRUCTURE ARY2[I,J] = [I*J](.ARY2+.I*J+.J)
OWN ARY2 X:Y:Z[10,10]
```

Itt E1=I*J, E2 elmarad és E3=.ARY2+.I*J+.J.

Ez a példa hatásában megegyezik az 1. példával. Ezen általános struktura deklaráció előnye, hogy a méretre vonatkozó adatok csak a társzegmens deklarációban szerepelnek, tehát ugyanaz a struktura definició felhasználható indexenként különböző méretü szegmensek deklarációjához. Figyeljük meg a 2. példában az I,J formális paraméterek és az E1 kifejezésben szereplő I,J nevek eltérő szerepét. A formális paraméterek az elérési algoritmusban csak . operátorral fordulhatnak elő és a hivatkozásban szereplő aktuális paraméterekkel helyettesitődnek. Az E1-

ben szereplő nevek pedig . nélkül szereplnek E3-ban, a méretre vonatozó értékeket közvetitik és a társzegmens deklarációjában szereplő értékekkel helyettesitődnek.

Mint a 2. példából is kitünik, a MAP deklaráció elmaradhat és helyette az OWN adja meg a szükséges információt.

4.2.1.2 A BLISS vezérlési szerkezetei

A BLISS kifejezés nyelv nemcsak olyan értelemben, hogy az értékadás is operátornak számit és ilymódon az értékadóutasitás egyszerüen kifejezéssé válik, hanem olyan értelemben is, hogy minden végrehajtható programelemnek van értéke és van hatása. Még a vezérlésátadó szerkezeteknek is van értékük, a konkrét szerkezeteknél majd látni fogjuk, hogy mi.

Kifejezésekkel dolgozva nem triviális, hogy a szekvencia mint vezérlési szerkezet jelen van a nyelvben. Pedig igy van. Egymásután következő ;-vel elválasztott kifejezések egymásután kiértékelődnek és a következő kifejezés kiértékelésének megkezdésekor (; előfordulása) az előző kifejezés értéke elvész. BEGIN ... END közé tett kifejezés sorozatot zárt kifejezés sorozatnak fogjuk nevezni.

Feltételes szerkezetek feltételeiben a bitminta legjobboldalibb bitje szolgáltatja a boolean értéket, mégpedig 1=TRUE, 0=FALSE. A kifejezések mindenütt lehetnek összetett kifejezések is.

Feltételes kifejezés

Formája: IF E1 THEN E2 ELSE E3, itt E1, E2 és E3 kifejezések. E1-től függően E2 vagy E3 kerül kiértékelésre és ennek értéke lesz a feltételes kifejezés értéke is. Az ELSE E3 elhagyható, E3 default értéke = 0.

Ciklus szerkezetek

A WHILE-szerkezetnek a következő formái használhatók a nyelvben:

WHILE E1 DO E	vizsgálat elől, ismétlés TRUE-ra
DO E WHILE E1	vizsgálat hátul, ismétlés TRUE-ra
UNTIL E1 DO E	vizsgálat elől, ismétlés FALSE-ra
DO E UNTIL E1	vizsgálat hátul, ismétlés FALSE-ra.

A ciklusváltozóval rendelkező ciklusok formája:

INCR NÉV FROM E1 TO E2 BY E3 DO E;

és DECR NÉV FROM E1 TO E2 BY E3 DO E;

E1, E2, E3 és E kifejezések (zárt kifejezés sorozat). E1, E2 és E3 kiértékelése a ciklusban való belépéskor történik INCR és DECR a lépésirányt jelenti. Minden ciklus kifejezés értéke = -1.

Esetszétválasztó kifejezések

CASE e OF SET E0; E1; ...; En-1; En TES

és SELECT e OF NSET E0:E1; E2:E3;...; E2n:E2n+1 TESN

A CASE kifejezés értéke Ee. A kifejezés értéke meghatározatlan, ha $0 \leq e \leq n$ nem teljesül.

A SELECT kifejezés végrehajtása a következőképpen történik: e kiértékelése, az E2i ($0 \leq i \leq n$) kifejezések kiértékelése, az E2i+1 kifejezések kiértékelése minden olyan i-re, amelyre e=E2i. Ha nincs ilyen i, akkor a SELECT kifejezés értéke = -1. Ha egy vagy több ilyen i van, akkor a legnagyobb ilyen i-hez tartozó E2i+1 lesz a SELECT kifejezés értéke.

Feltétel nélküli vezérlésátadás a nyelvben nincs.

A szökéskifejezéseket 3.3.5 alatt tárgyaltuk.

A függvényhivás kifejezés formája:

E(E1,E2,...,En)

Itt E egy olyan kifejezés, amely a hivandó szubprogram kódját tartalmazó szegmens nevét szolgáltatja. E1,E2,...,En az aktuális paraméterek, amelyek a függvénydeklarációban megnevezett formális paramétereket inicializálják. A paraméterátadás érték szerint történik. A függvényhivás-kifejezés értéke a függvénytörzs kiértékelésekor keletkező érték. Érdekesség, hogy E-nak nem kell explicite egy függvénynévnek lenni, de kiértékeléskor egy szegmensnévnek kell keletkezni.

Példa:

(CASE .X OF SET P1;P2;P3 TES) (.Z)

A korutin kifejezés fogalmát 3.3.7-ben részletesen tárgyaltuk. Kiegészitésként elmondjuk, hogy a CREATE utasitás pontos formája:

CREATE E (P1,P2,...,Pn) AT E2 LENGTH E3 THEN E4

Itt E kifejezés egy függvényt nevez meg, P1, P2,..., Pn az aktuális paraméterek, E2 és E3 a lefoglalt terület kezdőcime és hossza, E4 pedig a zárótevékenységet megadó kifejezés. E4 csak akkor értékelődik ki, ha a korutinból a törzs végén lépünk ki. A CREATE utasitás végrehajtása után a függvény korutinként viselkedik.

4.2.1.3 A BLISS programszerkezete

A BLISS program alapeleme a kifejezés. A zárt kifejezéssorozat más nyelvek összetett utasitásának felel meg, BEGIN ... END utasitászárójelpár között kifejezések pontosvesszővel elválasztott sorozata. Deklarációkat nem tartalmazhat. Felhasználása a vezérlési szerkezetekben történik. A blokk BEGIN ... END kulcsszavak között deklarációk és utasitások sorozata. A BLISS nyelv blokk fogalma scope szempontjából átmenetet képez a nyitott és a zárt scope között. Ugyanis a deklarációk által bevezetett azonositók lokálisak a blokkban, amelyben deklarálták őket és a nyitott scope szabályai szerint viselkednek, ugyanakkor a GLOBAL és EXTERNAL változók a zárt scope DEFINE-LIST és USE-LIST-jének szerepét töltik be.

Függvény

A függvény deklarációjáról a rendelkezésünkre álló irodalom nagyon keveset árul el. Valószinüleg igy néz ki:

FUNCTION név (P1,P2,...)=E

Itt P1,P2,... a formális paraméterek, E pedig a függvénytörzset megadó kifejezés. Deklarációkor a programkód tárolásához szükséges méretü társzegmens foglalódik a függvény számára és hely foglalódik a formális paraméterek számára is, amelyek funkcionálisan a lokális változókkal equivalesek, csak híváskor az aktuális paraméterek értékeit kapják kezdőértékül. Egy formális paraméter azonositója is tulajdonképpen a paraméter cimét reprezentálja, tehát értékére a . operátorral hivatkozhatunk. Rekurzió megengedett.

Irodalmunkból nem derült ki, hogy a részenkénti forditás hogyan történik, csupán rejtett célzás van arra, hogy van ilyen.

4.2.2 BCPL

A BCPL (Basic CPL) nyelvet M.Richards vezetésével a Cambridge-i Egyetemen fejlesztették ki 1969-ben compiler iró eszközként, de más rendszer-program irására is alkalmas. A nyelv jellemzője, hogy megkülönbözteti a jobbértéket (tartalom) és balértéket (cim) és a jobbmódu illetve balmódu kiértékelést. A kiértékelés módja környezettől függ, de operátorokkal is szabályozhatjuk. Erről 3.2.3-ban részletesebben szóltunk. Tipusfogalom nincs a nyelvben. Vezérlési szerkezetekben gazdag a nyelv, programszerkezete pedig az ALGOL-60-éhoz hasonló.

4.2.2.1 A BCPL nyelv adatszerkezete

A BCPL nyelv egyetlen adattipusa a bitminta. Bitmintával a programozó modellezheti az általa használni kivánt absztrakt adattipusokat. E célból sokféle operátor van beépitve a nyelvbe. Például a hagyományos aritmetikai operátorok alkalmazása révén a bitmintával az egész számokat lehet modellezni. Modellezhetünk igazságértékeket, karaktersorozatot stb. is. A jobb- és balérték megkülönböztetésével illetve a cimzési operátorok alkalmazásával egyszerü módszer áll rendelkezésünkre tömb és struktura kezeléséhez is.

A tömb ábrázolása legyen olyan, hogy a k méretü tömb számára foglaljunk le k db egymásutáni cellát, ugyanakkor a tömbnévhez rendeljünk mégegy cellát, aminek tartalma az első lefoglalt cella cime lesz, tehát a tömbnév tulajdonképpen a tömbre mutató pointer.

Ha V egy tömbnév és i egész szám, akkor

rv(V+i)

tulajdonképpen a tömb i-edik elemének tartalma. i természetesen változó, sőt kifejezés is lehet, az index változó szerepét tölti be. Mivel az ilyen tipusu kifejezések nagyon gyakoriak, ezért ezek rövid leirására egy uj szintaktikus jelölést vezettek be

rv(E1+E2) helyett E1↓E2 irható.

(rv=right value).

Az rv operátor a strukturák kezelésénél is hasznos. Tekintsük a következő kifejezést:

$$V\downarrow i \equiv rv(V+i) \qquad \text{(definició szerint)}$$
$$= rv(i+V) \qquad \text{(kommutativitás)}$$
$$= i\downarrow V$$

Ezek szerint $V\downarrow i$ és $i\downarrow V$ szemantikailag equivalens, azonban célszerü különbözőképpen értelmezni őket. Mint már láttuk, $V\downarrow i$ egy tömbhivatkozás. $i\downarrow V$ értelmezhető ugy, hogy i a szelektor, amit a V adatstrukturára alkalmazunk, tehát itt egy általános struktura-hivatkozásról van szó. Tetszőleges bonyolultságu összetett adatstrukturákat épithetünk fel, ha a sturturák elemei maguk is strukturák lehetnek.

A nyelvben szereplő konstansok a decimális, oktális vagy hexadecimális egész számok, stringek, karakter konstansok és a TRUE és FALSE logikai értékek, amelyek belső reprezentációja a csupa 1-es illetve csupa 0-ás bitsorozat.

A deklarációk a BCPL-ben specifikálják a bevezett név scope-ját, memória osztályát és kezdőértékét. A változó scope-ját az a szövegkörnyezet határozza meg, amelyben deklarálva volt pl. blokk, függvény vagy szubrutin törzs. Memória osztály szempontjából a változók 2 nagy osztályba sorolhatók: statikus és dinamikus változók. A statikus változók a program teljes futása során élnek, a dinamikus változók pedig a deklarációtól a scope-juk elhagyásáig. A következő deklaráció osztályok vannak: globális vektor, statikus változó, dinamikus vektor, dinamikus változó, függvény, szubrutin, cimke, formális paraméter, nevezett konstans. Ezek közül a globális vektor, statikus változó, függvény, szubrutin és cimke a statikus memória osztályba tartoznak, a dinamikus vektor, dinamikus változó és formális paraméter pedig a dinamikus memória osztályba. Nevezett konstans esetén nem történik helyfoglalás. Kezdőértékadás vektornak nem történhet, a formális paraméterek kezdőértékadása pedig híváskor történik, amikor az aktuális paraméterek értéke rendre bemásolódik a formális paraméterek számára foglalt cellákba.

A deklarációk formája a következő:

GLOBAL $(N1:K1; N2:K2,...$) - globális vektorok
N1,N2,... nevek; K1,K2,... konstans kifejezések =
= a vektor mérete - 1

STATIC $(N1=K1;N2=K2;... $) - statikus változók
N1,N2,...nevek; K1,K2,... konstans kifejezések =
= kezdőértékek

LET N=VEC K dinamikus vektor
N név, K konstans kifejezés = méret-1

LET N1,N2,... = E1,E2,... dinamikus változók
N1,N2,... nevek, E1,E2 kifejezések = kezdőértékek

LET N(P1,P2,...)=E függvény
N név; P1,P2,... formális paraméterek, E kifejezés a függvény törzse

LET N(P1,P2,...) BE C szubrutin
N név, P1,P2,... formális paraméterek, C utasitás

NÉV: cimke, előfordulása deklarálja

A formális paraméterek deklarálása a függvény illetve szubrutin deklarációban történik.

MANIFEST $(N1=K1;N2=K2;...$) nevezett konstansok
N1,N2,... nevek; K1,K2,... konstans kifejezések

A vektor, függvény, szubrutin és cimke deklaráció esetén a névhez egy statikus cella rendelődik, amely vektor esetén a O.elem cimét, függvény és szubrutin esetén a belépési cimet, cimke esetén a megcimkézett utasitás cimét tartalmazza.

Az adatokon végzett müveletek a hagyományos aritmetikai müveletek, bitenkénti logikai müveletek, shift müveletek és relációk. Mivel tipus nincs a nyelvben, igy tipus ellenőrzés és konverzió sincs.

4.2.2.2 A BCPL nyelv vezérlési szerkezetei

A BCPL vezérlési szerkezetekben igen gazdag. Több változata van a feltételes elágaztatásnak és a ciklusutasitásnak, van esetszétválasztó utasitás és vannak szökésutasitások. Érdekes jelenség a VALOF kifejezés, ami a különben élesen elválasztott kifejezésfogalom és utasitásfogalom ötvözete. A vezérlésátadá-

si szerkezetekben a feltétel lehet reláció, de lehet bármely kifejezés, amely logikai értéket (csupa 1 vagy csupa 0 bitsorozat) eredményez. Ha a feltételként szereplő kifejezés értéke se nem TRUE se nem FALSE, akkor a vezérlésátadási szerkezet hatása definiálatlan.

Feltételes vezérlésátadások

Formái:
```
IF   F  THEN  U1;
UNLESS F  THEN  U2;
TEST  F  THEN  U1  OR  U2;
```
Itt F feltétel U1 és U2 utasitás (összetett). Ha F=TRUE, akkor az U1 utasitás, ha F=FALSE, akkor az U2 utasitás kerül végrehajtásra. Tehát ez a nyelv az egymásbaágyazott feltételes utasitásoknál felmerülő kétértelmüséget a rövid és hosszu alak más elnevezésével oldotta fel.

Ciklusutasitások

A WHILE utasitások formái:
```
WHILE  F  DO  U;
UNTIL  F  DO  U;
U  REPEATWHILE  F;
U  REPEATUNTUL  F;
U  REPEAT;
```
Itt F feltétel U utasitás (általában összetett).
WHILE kulcsszó esetén a feltétel TRUE értékére, UNTIL kulcsszó esetén pedig FALSE értékre történik ismétlés. Mindkettőnél van lehetőség előirni, hogy a vizsgálat a törzs végrehajtása előtt vagy után történjen. A végrehajtás a felirásnak megfelelően történik. A legutolsó formánál nincs feltétel, a ciklusból való kiugrást más módon kell megoldani (pl. GOTO vagy BREAK).

A FOR utasitás formája:
```
FOR  N=K1 TO K2 BY K3 DO U;
```
Itt N név, K1 és K2 kifejezés, K3 konstans kifejezés, U utasitás. BY K3 elmaradhat, K3 default értéke = 1. K1 és K2 kifejezések kiértékelése a ciklusba való belépéskor történik.

Esetszétválasztó utasitás

Formája: SWITCHON K INTO U

Itt K kifejezés, U (összetett) utasitás.

U tartalmaz CASE K: vagy DEFAULT: alaku cimkéket, ahova a vezérlés adódik K értékétől függően (K itt konstans kifejezés). Az utasitások egymásutáni végrehajtása addig tart, mig egy ENDCASE utasitást nem találunk. Ennek hatására a vezérlés kilép az utasitásból. Tehát az esetek leirása tartalmazhatja egymást, "ráfolyhat" egymásra.

Feltétel nélküli vezérlésátadás

Formája: GOTO K, ahol K kifejezés.

K értékét mint cimet értelmezi és átadja a vezérlést erre a cimre. Egyébként a cimkék read-only statikus változóknak tekintődnek. Deklaráció előtt nem állhat cimke és egy utasitásnak több cimkéje is lehet.

Szökés utasitások

A BREAK, ENDCASE és FINISH utasitás hatására a vezérlés kilép rendre az őt tartalmazó legszükebb ciklusutasitásból, esetszétválasztó utasitásból illetve magából a programból.

Rutin-hivás és visszatérés

A függvény és a szubrutin hivás is a következő alaku:

K(P1,P2,...)

Itt K a belépési cimet megadó kifejezés, P1,P2,... az aktuális paraméterek. Mivel a rutinnév is egy statikus változót reprezentál, ezért a fentinek speciális esete a

NÉV(P1,P2,...)

alaku hivás. A paraméterátadás érték szerint történik. A függvényhivásnak van visszatérő értéke ezért kifejezés része lehet. A szubrutinhivásnak csak hatása van, ez önálló utasitás. Ha összecseréljük és szubrutint hivunk meg egy kifejezés részeként, az eredmény definiálatlan. Utasitásként azoban függvény is meghivható. A függvényből való visszatérés a függvénytörzset alkotó kifejezés kiértékelése után történik. A szubrutinból való visszatérés a RETURN utasitás hatására történik.

VALOF kifejezés

Formája: VALOF U,

ahol U összetett utasitás vagy blokk. A VALOF kifejezés kiértékelése ugy történik, hogy végrehajtjuk U utasitásait, amig a vezérlés egy RESULT IS utasitásra nem kerül, ennek argumentuma

lesz a VALOF kifejezés értéke. A RESULT IS utasitás hatására a vezérlés kilép az őt tartalmazó legszükebb VALOF blokkból.

4.2.2.3 A BCPL nyelv programszerkezete

A BCPL nyelv programszerkezete az ALGOL-60-éhoz hasonló. Egy lényeges eltérés, hogy a hatékony tárgykódgenerálás érdekében egy korlátozást vezettek be. Egy eljárás belsejében használt, de nem ott deklarált változóknak statikusnak kell lenni.

A BCPL nyelv utasitásorientált.

Az összetett utasitás utasitásoknak blokkzárójelekbe tett sorozata. A blokkzárójelek: $(és $). Szerepe a vezérlési szerkezetekben van, egyetlen utasitásnak számit. A változók scope-ját nem befolyásolja, kivéve a FOR ciklus, a VALOF utasitás és a szubrutin törzsét.

A blokk deklarációk, majd utasitások sorozata, amelyet blokkzárójelbe teszünk. A deklarált nevek scope-ja a deklarációtól a blokk végéig terjed. Az R10-es reprezentációban legfeljebb 31 blokkot skatulyázhatunk egymásba. Ilyen szempontból blokknak számit a VALOF kifejezés törzse, FOR ciklus magja, függvény és szubrutin törzse is.

A függvény egyetlen kifejezés kiszámitása, benne semmiféle lokális deklaráció nincs, kivéve a formális paramétereket, amelyek scope-ja csak erre a kifejezésre terjed ki.

A szubrutin törzse egyetlen utasitás, ami blokk is lehet és benne bármilyen deklaráció. Mind a függvény, mind a szubrutin lehet rekurziv. A függvény- és rutintörzs scope szempontjából átmenetet képez nyitott és zárt scope között. Nem nyitott scope, mert a beágyazó szubrutinban deklarált dinamikus változókra nem hivatkozhatunk a törzsön belül. Ugyanakkor nem is zárt scope, mert a globális változókra vagy a külsőbb egységekben deklarált statikus változókra lehet hivatkozni. Ezáltal a 3.1.7-ben tárgyalt tárkezelési mechanizmusok közül a 3. ábrán látható gazdaságos mechanizmust lehet megvalósitani.

Részenkénti fordithatóság

Egy BCPL programot nem szükséges egyetlen darabban forditanunk. A program külön forditott szegmensei közötti kommunikációra a globális vektor szolgál. A szegmensek összeszerkeszté-

sét a futtató computer operációs rendszere végzi, tehát ez gépfüggő.

4.3 Középszintü tipusos nyelvek

Erre a nyelvcsaládra az jellemző, hogy kompromisszumot köt a magasszintü programozási lehetőségek és a tárgykód hatékonysága között bizonyos foku portabilitás fenntartása mellett. Ezek a nyelvek nem gépközeliek, nem gépfüggőek és nem átlátszóak, de a hatékonyság fenntartása érdekében a nyelvi eszközöket is bizonyos fokig korlátozni kell pl. dinamikus tömbindex nem megengedett stb. Véleményem szerint ezek felelnek meg legjobban a SIL nyelvek követelményeinek.

4.3.1 XXPL

Az XXPL nyelvet az XPL nyelvből fejlesztették ki a Távközlési Kutató Intézet munkatársai. Az XPL nyelv az IBM 360 gép portabilis rendszerprogramozási nyelve, ezt honositották és fejlesztették tovább a TAKI-ban ICL/4-50 számitógépen. A honositás boot-strap módszerrel történt. A nyelv a fejlesztők szerint igen jó hatásfokkal fordítható. Tipusos nyelv, adatstrukturája közepesen gazdag, a különböző tipusokat csak olyan mértékben vették be a nyelvbe, ami nem megy a jó hatásfoku tárgykód generálása rovására. Vezérlési strukturái és programszerkezete alkalmazkodott a strukturált programozás követelményeihez, de kisebb-nagyobb megszoritásokkal itt is találkozunk. A középszintü tipusos nyelvek közül az XXPL a legalacsonyabb szintü, itt vannak a legerősebb megszoritások. Azonban kétségtelenül magasabb szinvonalu, kényelmesebb a PL360-nál, mivel nem gépfüggő és adatstrukturái is sokkal gazdagabbak.

4.3.1.1 Az XXPL adatszerkezete

A nyelvben két beépitett alaptipus van, ezek a numerikus és a nem numerikus adatok. Numerikus adattipus a FIXED és a rövid bitsorozat, BIT(k) $1 \leq k \leq 32$. A FIXED tipusu adatok egy szóban,

mig a rövid bitsorozatok hosszuktól függően egy byte-ban, félszóban illetve szóban kerülnek elhelyezésre. A nem numerikus adatok a karaktersorozatok (string) és a hosszu (k > 32) vagy definiálatlan méretü bitsorozatok. Ezek lehetnek statikusak vagy dinamikusak. A statikus stringek fix hosszuságuak és helyfoglalásuk állandó. A dinamikus adatok hossza változó és helyfoglalásuk dinamikus.

Deklarációjuk:

Statikus: CHARACTER(k) és
BIT(k) $32 < k \leq 2048$

Dinamikus: CHARACTER és
BIT

A rajtuk végezhető müveletek szempontjából nincs különbség a statikus és dinamikus stringek között. Megjegyezzük, hogy a string tulajdonképpen összetett adattipus, array of character, azonban az XXPL-ben egy karakterből álló karakter tipus nincs, csak összetett alakja, a string szerepel, ami bizonyos szempontból alaptipusnak tekinthető.

A felsorolt alaptipusokból vonatkozó és összetett tipusokat lehet képezni.

A pointer tipusu változó az indirekt hozzáférést biztositja. Deklarációja a REF kulcsszóval történik. A mutatott adat tipusát deklarációkor rögziteni kell. Mindenféle tipusra mutathat pointer, de a REF REF-et a forditóprogram REF FIXED-ként kezeli.

Egydimenziós fix méretü tömb képezhető az alaptipusokból és a pointer tipusból. A tömb méretét az azonositót követően zárójelben kell megadni. Ha itt * szerpel, akkor a tömb méretét a kezdőértékadás határozza meg.

Rekord deklarálásakor a REKORD alapszót zárójeles listában követi a struktura leirása. A névvel ellátott mezők csak FIXED, CHARACTER(k) vagy BIT(k) tipusuak vagy ezekből képzett tömbök lehetnek, rekord vagy referencia tipusu mező nem megengedett. A rekord elemeire való hivatkozás minősitett névvel (rekordnév.mezőnév) történik.

Mindenféle változónak adható kezdőérték, amit az INITIAL kulcsszó vezet be. Egyszerü változónál a kezdőérték konstans,

tömbnél és rekordnál konstansokból álló zárójeles lista. Pointer tipusnál konstans (abszolut cim) vagy már deklarált változó neve.

A numerikus adatokon a szokásos aritmetikai müveletek, bitenkénti logikai müveletek, shift müveletek és relációk értelmezhetők. Stirngekre a szokásos relációk, karakter kiemelés, hossz meghatározás, szelet, farok kiemelés, konkatenáció alkalmazható. Pointer tipusra az átutalás (SET), az inkrementálás és dekrementálás (PLUS és MINUS) és a mutatott érték inkrementálása és dekrementálása (UP-BY és DOWN-BY) müveletek alkalmazhatók. A PLUS és MINUS müveleteknél a mutató mozgatása a referált mennyiség tipusának megfelelően történik.

LITERALLY kulcsszóval névvel ellátott konstansok deklarálhatók. A numerikus konstansokat binárisan, quatrálisan, oktálisan, hexadecimálisan és decimálisan lehet megadni, a stringeket ' jelek közé kell tenni.

4.3.1.2 Az XXPL vezérlési strukturái

Az XXPL-ben boolean tipus nincs, igy a vezérlésátadó strukturákban feltételként vagy reláció vagy aritmetikai kifejezés legalacsonyabb helyértékü bitje szolgál (1=TRUE). A vezérlési strukturák közül a ciklus és az esetszétválasztó utasitások az END zárókulcsszóval végződnek. Az IF utasitásnak viszont nincs zárókulcsszava.

A feltételes vezérlésátadás formái:

IF K1 THEN U1

vagy IF K1 THEN U1 ELSE U2

Itt K1 kifejezés, U1 és U2 egyetlen zárt utasitás. Több feltételes utasitás egymásbaágyazásakor a legmélyebben lévő ELSE a hozzá legközelebb eső THEN-hez tartozik.

Ciklus utasitások

A WHILE utasitás alakja:

DO WHILE K1; U1 END;

Itt K1 kifejezés, U1 a ciklustörzs, utasitássorozat. K1 kiértékelése U1 lefutása előtt történik, és K1 igaz értéke mellett kerül sor U1 végrehajtására.

A FOR utasitás formája:

DO N1 = K1 TO K2 BY K3; U1 END;

Itt N1 változónév, K1, K2, K3 kifejezések, U1 utasitássorozat, a ciklustörzs. K1, K2 és K3 kiszámitása a ciklusba való belépés előtt történik. BY K3 elmaradhat, default értéke = 1. A TO kulcsszó helyett UP-TO és DOWN-TO kulcsszó használható a ciklusváltozó mozgásirányának jelzésére.

Esetszétválasztás

Formája:

DO CASE K1; $U_o,U_1,...U_n$ END;

Itt K1 kifejezés, $U_o,U_1,...,U_n$ utasitások. K1 értékének megfelelő sorszámu utasitás hajtódik végre. Ha K1 értéke negativ vagy >n, akkor ellenőrizhetetlen ugrás történik.

Feltétel nélküli vezérlésátadás

GOTO cimke;

Eljáráshivás és visszatérés

Az eljáráshivás alakjai:

CALL NÉV(P1,P2,...);

és NÉV(P1,P2,...)

Itt NÉV az eljárás neve és P1, P2,... az aktuális paraméterek. Nem kötelező az eljárás összes paraméterének felsorolása, a meg nem nevezett paraméterek default értéke = 0. A második eljáráshivás esetén a hivott eljárásból való visszatérés értékkel történik. A visszatérés a RETURN utasitás hatására történik, amelynek argumentuma a visszatérő értéket megadó kifejezés.

4.3.1.3 XXPL programszerkezete

Összetett utasitás. Az egyszerü utasitások összetett utasitásokká csoportosithatók a DO...; END; zárójelpár használatával. Az összetett utasitás a változók scope-ját nem korlátozza.

Blokk nincs az XXPL nyelvben.

Eljárások. A program strukturáját a blokkstruktura helyett az egymásbaágyazott eljárások strukturája adja meg. Az eljárások törzse nyitott scope-ot alkot lokális, globális változókkal. Eljárások rekurziv hivása nem megengedett. Ugyanazt az eljárást hivhatjuk függvényként értékkel és szubrutinként érték

nélkül. Az eljárások formális paramétereit az eljárás deklarációjakor fel kell sorolni és azok egyéb attributumait az eljáráson belüli deklarációval kell megadni.
Részenkénti fodithatóság. A nyelv megengedi a főprogramon kivüli önálló eljárások használatát. A szegmensben használt, a főprogramban deklarált változókat EXTERNAL jelöléssel kell ellátni. Ezeknek a fordító nem foglal helyet csak nevüket beteszi az ESD (External Symbol Dictionary)-be. A külön forditott részek összekapcsolását szerkesztő végzi.

4.3.2 IMP

Az IMP nyelvet az "Edinburgh Regional Computing Centre"-ben fejlesztették ki, az első Language Manual 1970-ben jelent meg. Azóta több gépre implementálták, többek között az ICL 4/75-re és az IBM 370/158-ra. Magyarországon az Országos Tervhivatal Számitóközpontjában honositották és használják.

Nagyon erős ALGOL-60 befolyás látszik a nyelven. A nyelv jellemzője, hogy mind adatszerkezete, mind vezérlési strukturái igen választékosak, azonban nem mindig a legmodernebb nyelvi eszközöket tartalmazzák. Legjobb példa erre a többirányu elágaztatást megvalósitó switch, amelyet a modern nyelvekben szinte teljesen kiszoritott a CASE utasitás.

Minden kulcsszó % jellel kezdődik, igy a kulcsszavak nem fenntartott nevek.

4.3.2.1 Az IMP adatszerkezete

Az IMP adatai lehetnek: aritmetikai változó (szám), string, pointer, tömb és rekord. Tipus definiálás csak rekordformátum esetén lehetséges.

A számok lehetnek: byteinteger, shortinteger, integer, real és longreal, ezek rendre 1,2,4,4,8 byte-ot foglalnak. A számkonstansok lehetnek decimális, hexadecimális és bináris felirásuak. A string mindig hosszával megadott karaktersorozat és az XXPL-hez hasonlóan itt is alaptipusként kezelődik. Egy n hosszu string belső ábrázolása n+1 byte-ban történik, az első byte tartalmazza a string hosszát.

Példa string deklarálásra: %STRING(5) S,Z;

A tömb maximálisan 7 dimenziós lehet. A deklarációban alsó és felső indexhatárt kell megadni és ezek változók is lehetnek, tehát dinamikus tömbméret lehetséges.

Pl. %INTEGER ARRAY IN(1:K)

Hivatkozásban az index egész kifejezés lehet.

A rekord deklarálásához először a rekord strukturáját, a rekordformátumot kell definiálni. Ez tulajdonképpen a tipus definiálás csirája. A rekordformátum mezőtipus és mezőnév párokból álló lista. A rekord deklarálásakor egy már definiált rekordformátumra kell hivatkozni.

Pl. %RECORDFORMAT F (%INTEGER A,%STRING(8) S)
%RECORD R(F)

Egy rekord mezője szám, string, pointer, egydimenziós fixméretü tömb és rekord lehet. Rekord tipusu mezőnél a formátumot is meg kell adni és annak már definiáltnak kell lenni. Ez a kikötés kizárja a rekurziv rekordot. Ha a mező rekordra mutató pointer, akkor a formátumdefiniciót követő %RECORDSPEC utasitás specifikálja a hivatkozott rekord tipusát.

Példa:

```
%RECORDFORMAT F (%INTEGER I, %RECORDNAME J)
%RECORDFORMAT K (%REAL X,Y)
%RECORDSPEC   F_J(K)
```

A rekordokból álló tömb, azaz a %RECORDARRAY deklarálása a következőképpen történik.

Pl. %RECORDFORMAT F (%INTEGER A, %REALARRAY X(1:5))
%RECORDARRAY RA(1:100)(F)

A hivatkozás minősitett névvel történik értelemszerüen:

Pl. RA(100)_X(5)

A pointer változó tulajdonképpen nem változó csak név olyan értelemben, hogy nem foglalódik neki hely a memóriában, csak feljegyződik a szimbólum táblába.

Deklarációja is erre utal, a tipusnévhez NAME szót kell ragasztani.

Pl. %INTEGERNAME I,J,K

Felhasználása ott történik, ha ugyanarra a változóra különböző, de azonos tipusu nevekkel akarunk hivatkozni. Felhasználás

előtt mindig equivalenciába kell hozni egy deklarált változóval.

A deklarált változóknak lehet prefixük, ami a változó kezelésére vonatkozóan ad utasitást. Az IMP terminológia nem igy nevezi, de tulajdonképpen memória osztályokról van szó. A prefix lehet: %OWN, %CONST, %EXTERNAL, %ENTRINSIC.

OWN prefix esetén a rutinból való kilépés és visszatérés esetén visszakapja kilépéskori értékét (mint az ALGOL-60 own változója), ez felel meg a többi nyelv static memória osztályának.

A CONST kulcsszó konstansokat deklarál.

Az EXTERNAL kulcsszó a változót más rutinok számára is hozzáférhetővé, azaz globálissá teszi, ez a terminológia eltér a szokásostól.

OWN, CONST és EXTERNAL prefix esetén a változónak kezdőérték adható. Tömbök közül csak fix méretü egydimenziós tömbök kaphatnak kezdőértéket. A kezdőérték konstansok listája, ismétlási tényező megengedett. Rekord nem kaphat kezdőértéket. %ENTRINSIC prefix esetén a változó számára nem történik helyfoglalás, más rutinok %EXTERNAL változói között kell keresni.

A számokon a szokásos aritmetikai müveletek, bitenkénti logikai müveltek és shift müveletek hajthatók végre (az utóbbiak real-en nem). Aritmetikai müveleteknél tipus ellenőrzés van és automatikus szélesités (byte-ból integer, integerből real stb.), a csonkitást (realból integer) pedig utasitásra végzi (INT). A logikai müveleteknél 32 bitre egészit ki. Érdekes, hogy értékadás 4 féle lehetséges: =, ←, == és →.

= aritmetikai értékadás, tipus összeférhetetlenség esetén hibajelzés;

← értékadás csonkitással, ha a jobboldal hosszabb mint a baloldal, akkor letöri a felesleges részt;

== pointer változók értékadása, itt a jobboldal nem a baloldal értéke lesz, hanem helyette használható;

→ resolution string müvelet (ld. 3.2.6.5).

A string müveletek közül a konkatenáció és a resolution, a string alapfüggvények közül pedig a karakter kiemelés, hossz meghatározás és szelet kiemelés szerepel.

Az ugyanahhoz az adathoz való többszörös hozzáférésre egyik eszköz a pointer változó.

Pl. %INTEGERARRAY TABLE (1:10,1:10)
%INTEGERNAME BASE
BASE==TABLE(1,1)

Ezután TABLE(1,1) BASE néven hivatkozható.

Az un. "MAPPING" függvények segitségével ugyanazt az adatot különböző tipusuként is hivatkozhatjuk és összetett adatstrukturának más strukturát adhatunk.

Pl. %LONGREAL X
%INTEGERNAME I,J
I== INTEGER(ADDR(X))
J== INTEGER(ADDR(X)+4).

4.3.2.2 Az IMP vezérlési szerkezetei

A feltételes vezérlésátadásoknál feltételként reláció, vagy relációknak az AND és OR operátorral összekötött sorozata áll. A kiértékelés sorrendjét a zárójelezés dönti el.

Feltételes vezérlésátadás

Formája:

%IF F1 %THEN U1
%IF F1 %THEN U1 ELSE U2

Itt F1 feltétel, U1 és U2 pedig feltétel nélküli utasitás.

U1 lehet: - egyszerü feltétel nélküli utasitás (értékadás, rutin hivás, feltétel nélküli ugró utasitás)
- %START...%FINISH utasitás zárójelek közé zárt utasitás sorozat,
- %AND kulcsszóval összekapcsolt utasitás sorozat.

Példák:

%IF A>B %THEN PRINT (A,5,3)
%IF A=1 %OR B=1 %THEN A=3 %AND C=3
%IF A=0 %THEN A=1 %ELSE %START...%FINISH

Ha az %IF kulcsszót %UNLESS kulcsszóra cseréljük, akkor az utasitássorozat a feltétel hamis voltakor kerül végrehajtásra. Ha az utasitássorozat egyetlen utasitásból áll, akkor a feltétel elé kerülhet. pl. A=0 %IF A=B.

Ciklus utasitások

WHILE utasitás

Formája: %WHILE F1 %THEN U1

Itt F1 feltétel, U1 feltétel nélküli utasitás.

U1 lehet: - egyszerü feltétel nélküli utasitás,
- %AND kulcsszóval összekapcsolt utasitássorozat,
- %CYCLE...%REPEAT zárójelpár közé fogott utasitássorozat.

A fenti esetben a feltételvizsgálat a ciklustörzs lefutása előtt történik és a törzs lefutása a feltétel igaz voltakor következik be.

%UNTIL F1 %THEN U1

esetén először U1 végrehajtódik, majd F1 vizsgálata következik és F1 igaz voltakor ismétlődik U1 végrehajtása.

Ha U1 egyszerü feltétel nélküli utasitás, akkor előre hozható

U1 %WHILE F1.

Ennek értelmezése teljesen azonos az előzőkkel.

Megjegyezzük, hogy ez a terminológia ellentétes a megszokottal. UNTIL kulcsszó esetén általában a feltétel hamis voltára történik az ismétlés, a feltételvizsgálat és a ciklustörzs lefutásának sorrendjét pedig általában a felirás sorrendje szabja meg.

Ciklus utasitás ciklusváltozóval és szabályos ismétléssel

Formája: %CYCLE N1=K1,K2,K3 U1,U2... %REPEAT

Itt N1 integer változó, K1,K2,K3 a kezdőértéket, lépést és végértéket megadó egész kifejezések, Ui-k utasitások. A ciklusváltozó értékét a cikluson belül nem szabad változtatni.

A %CYCLE...%REPEAT ciklus alkalmazható feltétel vagy ciklusváltozó nélkül is. Ilyenkor a ciklusból való kiugrás %EXIT vagy feltétel nélküli vezérlésátadás segitségével történhet.

Esetszétválasztás:

Az esetszétválasztásra a switch szolgál, ezt 3.3.3-ban részletesen ismertettük, ezért itt nem térünk ki rá.

Feltétel nélküli vezérlésátadás

Formája: → cimke

A cimke azonositó vagy egész szám lehet (1-től 16383-ig) utána kettősponttal.

Pl. 27:

:

→ 27

Szökéskifejezés

Az %EXIT utasitás hatására a vezérlés elhagyja a ciklust és a %REPEAT utasitást követő utasitásra kerül, de a ciklus változó (ha van) pillanatnyi értéke megőrződik.

Rutin hivás és visszatérés

A rutin hivás alakja:

Nl(Pl,P2,...,Pn)

Itt Nl függvénynév, Pl,P2,...,Pn akutális paraméterek. Az érték szerint hivott paraméterek helyén kifejezés is állhat. Az aktuális paraméterek tipusának meg kell egyezni a formális paraméterek tipusával.

A rutinból való kilépés a %RETURN utasitásra vagy a rutin fizika végét is jelentő %END utasitásra történik.

Függvényeljárás esetén a függvényértéket meghatározó RESULT= <expr.> utasitás is a függvénytörzs elhagyását eredményezi.

4.3.2.3 Az IMP programszerkezete

Az összetett utasitások feltételes vezérlési szerkezetekben szerepelnek, mégpedig

%START ... %FINISH - IF szerkezetben

%CYCLE ... %REPEAT - WHILE szerkezetben

A változók scope-ját nem befolyásolják.

Blokk. Az ALGOL-60-ból jól ismert blokk struktura érvényes.

%BEGIN Dl,D2,...,Ul,U2,...%END

Itt Dl,D2,... deklarációk, Ul,U2,... utasitások.

A memória verem szervezésü, ez a blokk strukturát, a dinamikus tömbméretet és rekurziv függvényhivást is lehetővé teszi. A blokkok 10 mélységig lehetnek egymásba ágyazva. A blokk nyitott scope. A legkülső blokk, a főprogram megkülönböztetésül %ENDPROGRAM kulcsszóval végződik.

Eljárás és függvény

Az eljárásokat és függvényeket deklarálni kell. A deklaráció és specifikácóból és leirásból áll. A kettő elválhat egymástól. A specifikáció tartalmazza a nevet és a formális paraméterlistát. Formális paraméterlista a paraméterek tipusát és nevét tartalmazza. A leirás megismétli a specifikációt és leirja a törzset. A specifikációnak mindig meg kell előzni a hivást, a leirásnak nem.

Példa:

```
specifikáció: %ROUTINESPEC NAME(%REALNAME X, %INTEGER I)
leirás:       %ROUTINE NAME (%REALNAME X, %INTEGER I)
               deklarációk
               utasitások
               %RETURN
               utasitások
               %END
```

Függvénydeklarációban ROUTINE helyett <tipus> FN áll. Az eljárástörzs nyitott scope, a globális változók értékét az eljárás deklaráláskori környezete határozza meg és nem a hiváskori érték.

A paraméterátadás lehet érték szerinti és név szerinti. Érték szerinti paraméterátadás esetén a formális paraméter specifikációjában tipus és név szerepel, ilyen a fenti deklaráció 2. paramétere. Az aktuális paraméter ekkor a megfelelő tipusu kifejezés lehet.

Név szerinti paraméterátadásnál a tipusnévhez a NAME szó kapcsolódik, ezután következik a formális paraméter neve, mint a fenti deklaráció 1. paraméterénél. Ez esetben az aktuális paraméternek a megfelelő tipusu változónévnek kell lenni. Az ilyen paraméterre való hivatkozás a rutinon belül lényegében az aktuális paraméter által megnevezett nem lokális változóra való hivatkozás. A név szerint hivott paraméterek értéke a hivás pillanatában számitódik ki.

Tömböt és rekordot csak név szerint lehet átadni.

Megjegyezzük, hogy mivel az aktuális paraméterek értéke a hivás pillanatában számitódik ki, itt tulajdonképpen hivatkozás szerinti paraméterátadásról van szó, csupán az IMP terminológia

hivja ezt név szerinti paraméterátadásnak. Paraméterátadásnál szigoru tipusvizsgálat van. Eljárás és függvény is lehet paraméter. Az eljárások és függvények lehetnek rekurzivok.

Részenkénti fordithatóság

Egy IMP program részenként fordítható olymódon, hogy a külön leforditott részeket a könyvtárba visszük, futás közben az eljárások onnan hivják majd egymást.

Egy külön forditott részt az IMP file-nak nevez. Ez a következőképpen néz ki:

- globális változók %OWN vagy %CONST prefix-szel,
- rutin leirások %EXTERNAL prefix-szel,

%END OF FILE

Több file fordítható le ilymódon és ezeket a könyvtárba kell vinni.

A főprogram a következőképpen néz ki:

```
%BEGIN
%EXTERNALROUTINSPEC utasitással specifikáljuk a használni
                    kivánt rutinokat, azaz megadjuk a nevét és
                    a formális paraméterlistát.
%EXTRINSIC kulcsszó után felsoroljuk a másutt deklarált
                    globális változókat.
deklarációk
   :
utasitások
   :
%END OF PROGRAM
```

4.3.3 C-nyelv

A C-nyelvet Ausztriában Murray Hill-ben a Bell Telephone Laboratories-ban fejlesztették ki 1973-74-ben a korábban ugyanitt létrehozott B-nyelv továbbfejlesztéseképpen. Implementálva van a PDP-11 UNIX operációs rendszerében, a HIS 6070 computeren és az IBM 360/370 gépcsaládon. A nyelv jellemzője az operátorokban való gazdaság, a pointer tipusu változó nagyfoku kiépi-

tettsége és a deklarációs lehetőségek széles választéka. A C-nyelv kifejezésnyelv.

4.3.3.1 A C-nyelv adatszerkezete

A nyelvben 4 alaptipus van: karakter, integer, szimpla és dupla-pontosságu real szám. Ezekből az alaptipusokból pointereket, tömböket és strukturákat lehet létrehozni. A nyelv megkülönbözteti az objektum, az érték és a balérték fogalmát. Ebből a szempontból a BCPL nyelvre hasonlit. Az objektum a memória hely, a balérték a cimke és az érték a tartalma. Balérték egy olyan kifejezés is, ami egy objektumra mutat, pl. egy azonosító. Vannak operátorok, amelyek balértéket hoznak létre, pl. a * operátor. Ha E egy pointer tipusu kifejezés, akkor *E egy balérték kifejezés, amely arra az objektumra hivatkozik, amelyre E mutat (indirektség). Az értékadás baloldalán mindig balértéknek kell állni.

A pointer tipus fontos jellemzője, hogy a tipus definicióba az is beletartozik, hogy milyen tipusu objektumra mutat. Mutathat adatra vagy függvényre. A függvények osztályozása ebből a szempontból a visszatérő érték tipusa szerint történik, pl. pointer to a function returning integer. A pointer belső ábrázolása az integerrel azonos és egy objektum cimét tartalmazza. Az arimetikai müveletek végezhetők rajtuk olymódon, hogy a pointer tartalma mindig kvantáltan változik. A kvantum a mutatott objektum hossza. Pl. ha a pointert növeljük n-nel, akkor a tartalma n* a mutatott rekord hosszával nő.

Tömböket lehet képezni az alaptipusokból, pointerből, tömbből és strukturából; függvényekből nem, de függvényekre mutató pointerekből igen. A méretet konstans kifejezéssel kell megadni. A tömb elemeire indexelt változókkal hivatkozunk.Pl. A[I]. Az indexelés 0-val kezdődik, az index kifejezésnek egész tipusunak kell lenni. A tömb belső ábrázolása olyan, hogy a legjobboldalibb index változik leggyorsabban. A C-nyelv jellegzetessége, hogy a tömbnév a tömb 0. elemének cimét reprezentálja, tehát tulajdonképpen a tömbre mutató pointer. Ezáltal hatékonyan tudunk hivatkozni a tömb elemeire. Azonban a tömb név csak read-only pointer.

Struktura mezője bármilyen adat lehet, természetesen saját maga nem, csak magára mutató pointer. Megkülönböztetjük a struktura leirást és a konkrét példányt, mindkettőnek külön neve van. A konkrét példány elemére kétféleképpen is hivatkozhatunk.

Példánynév . mezőnév

pointernév → mezőnév

Az utóbbinál a fordító a pointer által mutatott objektumot egy olyan szerkezetü strukturának feltételezi, aminek a mezőnév része. A struktura elemei a deklarálás sorrendjében helyezkednek el a memóriában a szóhatárokat figyelembe véve.

A nyelvben integer, karakter, real és string konstans van. A string tipusa "array of char", és a belső ábrázolásban nullabyte zárja le.

A deklaráció megadja a változó memória osztályát, tipusát és nevét, esetleg kezdőértékét. A memória osztály lehet: AUTO, STATIC, EXTERN és REGISTER. Ezek a 3.2.9 beli terminológiának megfelelően a lokális, statikus, external és regiszter osztályok.

A tipus megadására az INT, CHAR, FLOAT, DOUBLE és STRUCT kulcsszavakat használhatjuk. Struktura deklarálásnál 3 lehetőség van:

- definiálunk egy strukturát és deklaráljuk a konkrét példányokat,
- definiálunk egy strukturát, nevet adunk neki (structure tag) és deklarálhatunk konkrét példányokat,
- egy előzőleg definiált struktura nevére hivatkozva deklaráljuk a konkrét példányokat.

```
Példa: STRUCT tnode {CHAR tword [20]; INT count;
           STRUCT tnode   *left;};
       STRUCT tnode S, *Sp;
```

A pointer, függvény és tömb tipus deklarálása a deklarált azonositó tipusát módositó jelölések segitségével történik, ezekből kifejezéseket lehet felépiteni.

Pointer dekl:	T *D	D tipusa: pointer to T
Függvény dekl:	T D()	D tipusa: function returning T
Tömb dekl:	T D[Kl]	D tipusa: Kl méretü tömb T tipusu elemekből
	T D[]	D tipusa: határozatlan méretü tömb

Példa:
Az INT i, * ip, f(), * fip(), (* pfi)(); deklarációban

i integer,
ip pointer to integer,
f function returning integer,
fip function returning a pointer to an integer,
pfi pointer to a function which returns an integer.

Kezdőérték rutinban deklarált változónak nem adható, csak a minden rutinon kivül, globálisan deklarált változónak. A kezdőérték konstans kifejezés vagy ezek listája. Tipus egyezésnek kell lenni. Tömb kezdőértékadásnál a tömb mérete a listában szereplő kifejezések számának és a deklarált méretnek a maximuma. A struktura kezdőértékadása gépfüggő.
A C-nyelv nagyon gazdag operátorokban, a szokástól eltérő operátorok a hatékony tárgykódgenerálást teszik lehetővé. A primary-expression operátorok:

()	[]	.	→
függvényhivás	indexelés	mezőszelektor rekordból	mezőszelektor a mutatott rekordból

Az unáris operátorok:

*	indirektség	++ increment	alaptipus esetén
&	cimmeghatározás	-- decrement	1-gyel,
-	előjelforditás		pointer esetén a
!	predikátum:		mutatott tipus egy
	IF expr = 0 THEN 1,		elemének hosszával
	ELSE 0		
~	1-es komponens	sizeof: az operandus byte-okban mért hossza	

A bináris operátorok a szokásos aritmetikai, bitenkénti logikai és shift müveletek és a relációk. Az operátorok tipus függőek, a konverzió fajtái pontosan meg vannak határozva. Érdekesség, hogy pointer és integer közötti müveleteknél az integer értéke szorzódik a pointált tipus egy elemének hosszával.
A következő predikátumok vannak a nyelvben:

E1 && E2	IF E1 ≠ 0 AND E2 ≠ 0 THEN 1 ELSE 0
E1 || E2	IF E1 ≠ 0 OR E2 ≠ 0 THEN 1 ELSE 0
E1? E2:E3	IF E1 ≠ 0 THEN E2 ELSE E3

Az értékadás szokásos formáján (balérték=expression) kivül olyan értékadó operátorok is vannak, amelyek az értékadás és a bináris operátorok kombinációi. Ezek =+ =- =* =/ =% =>> =<< =& =^ =|

Jelentésük: E1 = op E2 equivalens E1 = E1 op E2

4.3.3.2 A C-nyelv vezérlési szerkezetei

A C-nyelvre jellemzők a modern vezérlési szerkezetek, de azért van GOTO is. A feltételes szerkezetekben a feltételt aritmetikai kifejezések 0 vagy nem 0 értéke jelenti. Mint már kitünt, boolean változó nincs a nyelvben, de relációkból logikai operátorokkal összekapcsolt logikai kifejezés sincs, mint pl. az IMP-ben. Helyette a relációk is számot eredményeznek. Értékük = 1, ha a reláció igaz, és 0, ha hamis. A vezérlési szerkezetekre jellemző, hogy nem záró kulcsszó, hanem összetett utasitás van bennük.

Feltételes vezérlésátadás

Formája: IF (E) U1

vagy IF (E) U1 ELSE U2

Itt E expression, U1 és U2 utasitások.

Tehát a feltételt THEN helyett zárójel zárja.

Ha E ≠ 0, akkor U1 utasitás kerül végrehajtásra. Ha E = 0, akkor az első esetben az IF utasitást követő utasitás, a második esetben pedig U2. Egymásbaágyazott IF utasitások esetén az ELSE mindig a legutoljára előfordult ELSE nélküli IF-hez tartozik.

Ciklus utasitások

WHILE utasitás

Formái: WHILE (E) U

E - kifejezés, U - utasitás

Feltételvizsgálat a törzs lefutása előtt.

DO U WHILE (E);

Feltételvizsgálat a törzs lefutása után.

Mindkét esetben ha E ≠ 0, akkor U végrehajtódik.

FOR utasitás

Formája: FOR (E1; E2; E3) U

Jelentése: E1;

WHILE (E2) {U, E3}

E1 kiértékelődik, ez kell hogy tartalmazza a kezdőérték beállitást a ciklus változó számára; E2 a feltétel, a ciklus befejeződik, ha E2 nullává válik; U a ciklus törzsét alkotó, rendszerint összetett utasitás; E3 pedig a ciklusváltozó megfelelő módositása mindenegyes végrehajtás után. E1, E2 és E3 közül mindegyiket elhagyhatjuk. E2 default értéke = 1, E1 és E3-é pedig az üres utasitás.

Esetszétválasztás

A többirányu elágaztatásra a SWITCH utasitás szolgál. A szóhasználat megtévesztő, itt nem switch-ről, hanem a CASE utasitásnak megfelelő esetszétválasztó utasitásról van szó.

Formája: SWITCH (E) U

Az E szelektor kifejezés tipusa int vagy char. U összetett utasitás. Benne minden utasitás viselhet CASE-prefix-szel ellátott cimkét, aminek formája CASE K:, ahol K int vagy char konstans kifejezés. Egy SWITCH utasitáson belül nem lehet két ugyanolyan értékü cimke. Ezenkivül szerepelhet a DEFAULT: cimke is. Az egyes eseteknek csak a belépési pontja van meghatározva, a kilépésről a programozónak utasitással kell gondoskodnia (BREAK vagy GOTO utasitás). Ha nincsen az E értékével megegyező cimke, akkor a DEFAULT: cimkére adódik a vezérlés, ha olyan sincs, akkor az utasitás üres utasitásnak tekintődik.

Feltétel nélküli vezérlésátadás

Formája: GOTO K

Itt K egy pointer to int tipusu kifejezés, amelyik egy cimkét határoz meg. A cimkéknek nincs saját tipusuk, hanem array of int tipusuak. A label-változókat pointer to int tipusunak kell deklarálni. GOTO label-változó utasitást csak akkor adhatunk ki, ha a változó előzőleg egy cimkét kapott értékül.

Szökéskifejezések

A BREAK-utasitás a legkisebb beágyazó WHILE, FOR vagy SWITCH utasitásból való kilépést okoz, a vezérlés az elhagyott szerke-

zetet követő utasitásra adódik.
A CONTINUE utasitás a WHILE vagy FOR ciklus ismétlődési pontjára adja a vezérlést.
Függvény hivás és visszatérés
Formája: NÉV(P1,P2,...)
Itt P1,P2,... az aktuális paraméterek.

A paraméterek tipusa az alaptipusok és pointer lehet. A tömb automatikusan pointer-nek tekintődik, struktura és függvény nem lehet paraméter, csak ezekre mutató pointer. A paraméterátadás érték szerint történik.
A RETURN(E)
vagy RETURN
utasitás a függvényből értékkel vagy anélkül való visszatérésre szolgál. A függvény fizikai végének elérése érték nélkül való visszatérést okoz.

4.3.3.3 A C-nyelv programszerkezete

Expression statement. A C-nyelv kifejezés orientált nyelv. Az adatokon vérehajtandó tevékenységet kifejezésekkel irja le, az értékadás is kifejezés. Két fontos expression statement tipus van: az értékadás és a függvényhivás.

Összetett utasitás. Utasitás sorozat { } zárójelpár között. A vezérlési szerkezetekben van szerepük. A változók scope-ját nem befolyásolják.

Blokkstruktura nincs a nyelvben.

A függvény a legfontosabb programegység a nyelvben. Nincs külön eljárás és függvény, de a függvénynek lehet értéke és hatása is. Ha a függvény érték nélkül tér vissza, akkor nyilván a hatása miatt hivtuk, tehát eljárásnak tekinthető. A függvénytörzs nyitott scope.

Függvényeket nem lehet egymásba ágyazottan deklarálni. Igy tulajdonképpen a változók scope-ja szempontjából mindössze két szint van, lokális változók a függvényeken belül és globális változók minden függvénydeklaráción kivül.

A program tulajdonképpen adat és függvény deklarációk sorozata. A deklarált függvények között van egy MAIN nevü. Ez kapja meg először a vezérlést, a függvények ezután hivhatják

egymást. A program részenként fordítható olymódon, hogy a deklaráció sorozatot alkalmasan széttagoljuk, pl. egy-egy hosszu függvénydeklarációt külön forditunk és könyvtárba vesszük. A könyvtár egy legkülsőbb szintként kerül a program fölé. Ha egy változó nincs a lokális változók között, akkor a globális változók között keresi a fordító, és ha ott sem találja, akkor a könyvtárban. Az egyszerre forditott részt file-nak hivja a C-nyelv és END OF FILE utasitással zárja le, mig az egész programot az END OF PROGRAM zárja.

4.3.4 GESAL

A GESAL-t az MTA SZTAKI-ban fejlesztették ki belső használatra 1975-76-ban. Azóta több rendszerprogram készült ezen a nyelven. A nyelv nem törekszik teljes portabilitásra, csak a 8 bites byte-okból álló memóriával rendelkező computerek számára portabilis. A nyelv jellemzője az adatszerkezet gazdagsága, a sokféle tipus és tipus definiálási lehetőség. A programszerkezet viszont egyszerü, blokkstruktura nincs, az eljárások deklarálása csak a fő szinten történhet, ez hatékony tárgykód generálást tesz lehetővé. A vezérlési szerkezetek megfelelnek a strukturált programozás követelményeinek.

4.3.4.1 A GESAL adatszerkezete

A GESAL nyelvben az adatoknak mindig van tipusa, az operátorok tipus függőek. Vannak beépitett és user-definiált tipusok. Az user-definiált tipusoknak is vannak osztályai, amelyeknek kulcsszavai fixek. A tipus definiálás kétféleképpen történhet:

1. MODE-deklarációnál nevet adunk a definiálandó tipusnak és azután a nevezett tipushoz deklarálhatunk változókat.
2. A tipus definiálás és a változó deklarálás egy utasitásban történik.

A nyelvben a következő adattipusok illetve tipus-osztályok vannak.

Beépitett tipusok: char, byte, int, longint, real.
Az user definiált tipusok osztályai: enumeration, set, reference, array, structure és union.
Az adattipusokon kivül procedure-tipus és label tipus is van.

Enumeráció tipus-osztály esetén a felhasználó adja meg a lehetséges értékek felsorolását, amelyek egyike lehet a változó értéke. Kulcsszava ENUM vagy ENUMB. Az ENUM tipus 1 szót foglal mig ENUMB 1 byte-ot.

Set tipus-osztály esetén a felhasználó megad egy névvel ellátott elemekből álló halmazt, hasonlóan mint az enumerációnál. A halmaz elemeinek egy-egy bit felel meg a belső ábrázolásnál. A set tipusu változó az elemek tetszőleges kombinációját veheti értékül. A set tipus-osztály kulcsszavai SET és SETB, rendre a szvas vagy byte-os helyfoglalásnak megfelelően. Tehát a set-tipus a GESAL-ban önálló és nem vonatkozó tipus, mivel nem egy korábban definiált enumeráció tipusra vonatkozik, hanem helyben definiálja az elemkészletét. A set-tipus mask-ként használható.

Referencia tipus-osztály esetén a felhasználó azt adja meg, hogy egy konkrét REF tipus milyen tipusra mutat. Mutathat adat, procedure vagy label tipusra. Kulcsszava: REF.

Tömb képezhető tetszőleges, de azonos tipusu adatokból. A tömb méretét [ ]-ben álló konstans-kifejezéssel kell megadni, [ ] implicit tömbméret meghatározást jelent. A több dimenziós tömböt tömbök tömbjeként kell megadni. A tömb elemeire való hivatkozás indexeléssel történik. A tömb első eleme 0 indexü.

A struktura mezői tetszőleges adattipusuak lehetnek. Kulcsszava: STRUCT. A struktura mezőire minősitett névvel hivatkozhatunk. A mezőnevek hatásköre csak a strukturára terjed ki.

Az union type arra szolgál, hogy ugyanahhoz az objektumhoz különböző tipusuként férhessünk hozzá. Szintaxisa a strukturáéhoz hasonló, de a mezők ugyanarra a területre kerülnek. Az union tipusu változó hossza a mezők hosszának maximuma. Az union tipus változóra a struct-hoz hasonlóan, minősitett névvel hivatkozunk. Kulcsszava: UNION.
Példa: UNION (INT I; REF REAL TREAL) X,Y;
hivatkozás: X.I, Y.TREAL

A label tipusu változó értéke mindig csak cimke lehet. Kulcsszava: LABVAR. A GOTO utasitás argumentuma lehet, pointer mutathat rá, tömb vagy struktura eleme lehet. A cimkék LABVAR tipusu konstansoknak minősülnek.

Konstansok. A nyelvben integer, real, karakter és string konstans van. Az integer lehet hexadecimális, oktális, bináris és decimális felirásu. A karakter és string konstans is karaktersorozat és array of character tipusu, azonban a karakter konstans annyi byte-ot foglal, ahány karakterből áll, a string konstans pedig eggyel többet és az utolsó helyen 0 zárókarakter áll. A string konstans '' jelek között áll, a karakter konstans pedig C betü után álló string konstans.

Deklarációk

Az adatdeklaráció a következőképpen történik:

<prefix><adattipus><név>=<kezdőérték>;

A prefix lehet CONST, STATIC, EXT, ENTRY vagy üres, ezek a konstansok deklarációnak illetve a statikus, external és globális memória osztályoknak felelnek meg.

Nevezett konstans BYTE, INT, LONGINT, REAL, CHAR és SET tipusu lehet. Az adattipus vagy a beépitett tipusok egyike, vagy egy már definiált adattipus neve, vagy egy adattipus leirása. A név a deklarálandó változó neve. A kezdőérték konstansokból képzett kifejezés vagy ezek zárójeles listája. Az utóbbi összetett adattipusok (tömb és struktura) kezdőértékadására szolgál, ilyenkor az adatszerkezetnek megfelelően kell zárójelezni. Implicit tömbméret meghatározásnál a kezdőértékek listája szabja meg a tömb méretét. Referencia tipusu változónak is adható kezdőérték.

Pl. INT I

REF INT R1 = LOC I

Kezdőérték csak globális és statikus memória osztályu változóknak adható.

A tipus definiálás formája:

MODE<név>=<tipusleirás>

A tipusleirás az ENUM, SET, STRUCT, REF és UNION kulcsszavak után a megfelelő leirás, tömb esetén pedig []-ben a tömbméret megadás. Tipus definiálásnál nem történik helyfoglalás.

A nyelvben az operátorok tipus-függőek. Az aritmetikai tipusokra alkalmazhatjuk a szokásos aritmetikai müveleteket, bitenkénti logikai müveleteket, shift müveleteket és a store müveletet, ezenkivül a relációkat, amelyek eredménye logikai érték. A többi tipusra a következő operátorok alkalmazhatók:

Enumeráció: relációk, store,

Set: + (UNION), OR, XOR, AND, NOT, IN, IS, store.

Reference: =, ≠ relációk, store.

Az ARRAY, STRUCTURE és UNION tipus csak elemein keresztül érhető el, azaz indexeléssel és minősitett névvel. Minden tipusra alkalmazható a LOC müvelet, amelynek eredménye az adatra mutató pointer. Pointer tipusra alkalmazott ↑ (dereferencing) operátor a referált adatot eredményezi.

4.3.4.2 A GESAL vezérlési szerkezetei

A feltételes szerkezetek feltételei relációkból AND és OR operátorokkal képzett kifejezések.

IF szerkezet

Formája: IF F1 THEN U1
ELSIF F2 THEN U2
:
ELSE Un
FI

F1,F2,...F_{n-1} feltételek, U1,U2,...Un utasitássorozatok. Az utasitássorozat tetszőleges utasitásokat, tehát ujabb IF szerkezetet is tartalmazhat, mivel a zárókulcsszó (FI) jelenléte egyértelmüen eldönti a szerkezet jelentését. Az ELSIF és ELSE ágak elhagyhatók.

Ciklus utasitások

A DO-szerkezet a szokásos WHILE és FOR utasitások összeolvasztása, azonban a szerkezet bármelyik része elhagyható.

A DO-utasitás formája: FOR N1 FROM K1
TO K2 BY K3 WHILE F1 DO U UNTIL F2 OD.

Itt: N1 - integer változónév, K1,K2,K3 integer kifejezések és F1,F2 feltételek, U utasitássorozat.
Az U utasitássorozat ismétlődik, ha a ciklus változó értéke a ciklus határok között van, F1 értéke TRUE és F2 értéke FALSE. A default értékek a következők: K1=0, K2=∞, K3=1, F1=TRUE és F2=FALSE. Ha valamelyik érték elmarad, akkor a megfelelő kulcsszót is el kell hagyni. TO helyett UPTO vagy DOWNTO használandó és a ciklusváltozó értéke ennek megfelelően változik.

CASE utasitás

Formája: CASE K1
```
      OF C11,C12,...: U1
               ⋮
      OF Cn1,Cn2,...: Un
      OUT             Un+1
      ESAC
```
Itt K1 a szelektor kifejezés; C11,C12,...Cn1,Cn2,... kifejezések az OF-cimkék; U1,...Un,Un+1 pedig utasitássorozatok. Az eseteket az OF kulcsszó ismétlése zárja le.

Egy esethez több cimke is tartozhat. Az esetek felsorolását az ESAC kulcsszó zárja. K1 kiértékelődik és rendre összehasonlitódik az OF-cimkékkel. Az első egyezésnél a cimkéhez tartozó utasitássorozat végrehajtódik. Ha nincs egyezés, akkor Un+1 kerül végrehajtásra, ha van, ha nincs OUT rész, akkor semmi sem történik.

Feltétel nélküli vezérlésátadás

Formája: GOTO LE

Az LE location expression lehet cimke vagy LABVAR tipusu változó, vagy ilyent eredményező kifejezés. Pl. LABVAR tömb eleme, vagy REF LABVAR tipusu változó dereferenciálása stb.

Eljárás hivás és visszatérés

Hivás: Név(P1,P2,...)

Itt Név a hivott eljárás neve, P1,P2,... pedig az aktuális paramétereknek megfelelő kifejezések. Ezek tipusának rendre meg kell egyezni a formális paraméterek tipusával. Az eljáráshivás lehet önálló utasitás, ha azonban van visszatérő paramétere, akkor értéket képvisel és kifejezés része lehet. Az eljárásból való visszatérésre nincs külön utasitás, az eljárás fizikai vé-

gének elérése az eljárásból való visszatérést okozza. A visszatérő érték a deklarációban kijelölt változó értéke.

4.3.4.3 A GESAL programszerkezete

A GESAL programszerkezete nagyon egyszerü. Sem összetett utasitás, sem blokk nincs benne. Az eljárásdeklarációk nem lehetnek egymásba ágyazva. A program tulajdonképpen deklarációk sorozata, amely deklarációk vegyesen tartalmaznak adat- és eljárásdeklarációkat. Az itt definiált adatok a globális adatok, amelyekre mindegyik eljárás hivatkozhat. Az eljárások közül van egy, a MAIN, a főprogram, ez inditja a többi eljárását, amelyek hivhatják egymást. Rekurziv hivás megengedett. Az eljárás egy olyan programegység, amely nyitott scope-ot képvisel. Az eljárás törzsén belül deklarált változók lokálisak az eljárásra nézve. Az eljárások osztályokba sorolhatók a formális paramétereik és a visszatérő érték milyensége szerint. Csak az azonos eljárásosztályra mutató pointerek helyettesithetők egymással. Az eljárásosztályok definiálása a tipusdefiniálás szabályai szerint történik. Tehát egy eljárás deklarálása kétféleképpen történhet. Definiálhatom előre az eljárásosztályt.

```
MODE  N1  = PROC (T1, T2, ... : TR)
```

Itt N1 az eljárásosztály neve, T1, T2, ... a formális paraméterek tipusa és TR a visszatérő érték tipusa. Ezután deklaráltak egy egy ebbe az osztályba tartozó eljárást.

```
N1   N2 = (F1, F2, ...: FR)
     BEGIN S1, S2, ... END
```

Itt N1 az eljárásosztály neve, N2 a konkrét eljárás neve, F1, F2, ... a formális input paraméterek neve, FR a visszatérő értéket tartalmazó változó neve, S1, S2, ... pedig vegyesen deklarációk és végrehajtó utasitások.

Az eljárásosztályt megadhatom az eljárás deklarációban is. Igy:

```
PROC (T1, T2, ... :TR) N2 = (F1, F2, ... :FR)  BEGIN ... END
```

A formális paraméterek csak egyszerü tipusuak lehetnek, összetett tipusu adatstruktura nem adható át, csak arra mutató pointer, viszont függvényre mutató pointer is átadható. A paraméterátadás érték szerint történik.

4.3.5 PASCAL

A PASCAL nyelvet N. Wirth fejlesztette ki azzal a céllal, hogy legyen egy nyelv, amely szisztematikusan épül fel néhány alapfogalom köré és amely alkalmas arra, hogy vele kezdjék a programozás tanitását. Ugyanakkor célul tüzték ki a hatékony implementálhatóságot is. Az első report 1971-ben jelent meg [15] amelyet 1973-ban követett az axiomatikus leirás [18], majd 1974-ben a Revised Report [19]. A nyelv több ujdonságot vezetett be, pl. az enumeráció, subrange és file tipusokat, a változó rekordot és a tipus definiálást. Ugyanakkor a hatékony implementáció érdekében elhagyott néhány már megszokott sajátságot, pl. a blokk-strukturát, a dinamikus tömbméretet, és a részenkénti fordithatóságot. Mindkét irányu változtatás heves birálatot váltott ki a kortársakból, sok cikk jelent meg pro és kontra pl. [17, 20], mig végül is a PASCAL fogalmai bevonultak a köztudatba. A fellángolt vitához az is nagyban hozzájárult, hogy az eredeti report, de még a Revised Report is tartalmaz pontatlan vagy egyenesen hibás megfogalmazásokat. Az eredmény azonban mégis az, hogy alig van néhány nyelv, amely olyan nagy hatást gyakorolt volna az utána keletkező nyelvekre, mint a PASCAL.

Jóllehet nem kifejezetten rendszerprogramozási nyelvnek szánták, gazdag adatstrukturái miatt erre a célra is alkalmas és nem gát a rossz hatásfoku tárgykód sem, hiszen az egyik cél éppen a hatékony implementáció volt.

4.3.5.1 A PASCAL adatszerkezete

A PASCAL tipusos nyelv, minden adatnak tipusa van. Az adattipus vagy közvetlenül a változó deklarációban van leirva vagy előre lehet definiálni és a változó deklaráláskor csak a tipusnévre hivatkozni. A tipus a PASCAL-ban lehet egyszerü, összetett és pointer tipus. Az egyszerü tipus lehet skalár és subrange.

A skalár tipus lényegében az enumeráció, ezenkivül a beépitett standard tipusok, INT, REAL, BOOLEAN és CHAR. A skalár tipus jellegzetessége, hogy értékkészlete rendezett. A subrange tipus egy skalár tipusra vonatkozóan van definiálva megadva az

értékkészlet alsó és felső határát. A subrange tipussal 3.2.5-ben részletesen foglalkoztunk. Az összetett tipus lehet pakolt vagy pakolatlan. A pakolt tipus esetén a változók memória beli elhelyezése a lehető leggazdaságosabb, mégha a futási idő rovására is.

Az összetett tipusok: tömb, rekord, set és file.

A tömb deklarációja az alaptipust és az index tipusát rögziti. Az alaptipus bármilyen tipus lehet. Az index tipusa egyszerü tipus lehet (nyilván a real kivételével), amelynek minden eleme lehet index. Nyilvánvalóan indextipusnak véges tipust, azaz enumerációt, subranget, BOOLEAN-t vagy CHAR-t kell kijelölni.

A rekord mezői tetszőleges tipusu adatok lehetnek. A PASCAL nyelv vezette be a változó rekord fogalmát. Erről 3.2.6-ban részletesen irtunk.

A set tipus értékkészlete az alaptipus értékkészletéből képzett részhalmazok. Az alaptipus nem lehet összetett tipus, sőt csak véges tipus lehet. A set tipus tulajdonképpen nem összetett, hanem vonatkozó tipus, mivel nem komponensekből áll, hanem az értékkészlet megadási módja különleges.

A file szekvenciális hozzáférésü, azonos tipusu elemekből álló adatsokaság. A file részletes leirását szintén 3.2.6 alatt találjuk.

A pointer tipus használata eltér a többi SIL nyelvben szokásos használattól. Dinamikus változók kezelésére szolgálnak. A dinamikus változók scope-ja program utasitás hatására változik és semmi korrelációban nincs a program strukturájával. Részletes leirásukat 3.2.5 alatt találjuk meg. A PASCAL-ban a pointeren semmiféle müvelet nem végezhető, kivéve az egyenlőség vizsgálatot.

Konstansok. Az integer konstansok csak decimálisan irhatók, a real-ek skálafaktorral, a string konstans packed array of char tipusu, a boolean konstansok a TRUE és FALSE logikai értékek, a pointer konstans pedig a NIL. A set konstans [ ]-ben felsorolva azok az elemek, amelyek a részhalmazba tartoznak. A konstansokhoz konstans definicióval nevet is rendelhetünk. Kulcsszava: CONST.

Mint már emlitettük, a tipus definició és a változó dekla-

ráció eltér egymástól.

A tipus definiálás formája:

<név> = <tipusleirás>

A tipusleirás formái:

az alap tipusok:	INTEGER|REAL|BOOLEAN|CHAR
enumeráció:	(N1,N2,...) N1,N2,... nevek,
subrange:	K1..K2 K1 és K2 konstansok,
tömb:	ARRAY [I1,I2,...] OF T
	I1,I2,... index-tipusok, T az alaptipus,
rekord:	RECORD M11,M12,...:T1;M21,...:T2;...END
	M11,M12,...M21,... nevek, T1,T2,... tipusok,
	a változó rekordról 3.2.6-ban,
set:	SET OF T; T tipus,
file:	FILE OF T, T tipus,
pointer:	↑TN
	TN tipusnév.

A változó deklarálás formája:

N1,N2,... : T

Itt N1,N2,... nevek, T tipus.

Memória osztály és kezdőértékadás a PASCAL-ban nincs.

Müveletek. A PASCAL nyelv szegény müveletekben. Az operátorok tipus függőek és precedencia csoportokba tartoznak. A szokásos aritmetikai, logikai és reláció müveletek végezhetők, valamint a set müveletek (set union, set-difference, intersection és IN reláció). Nincs bitenkénti logikai müvelet és shift müvelet aritmetikai tipusra, nincs müvelet a pointer tipusra. Egy kifejezés operandusa lehet indexelt változó, minősitett név, file buffer változó és pointált változó is (pointer↑).

4.3.5.2 A PASCAL vezérlési strukturái

Mivel a nyelvben szerepel BOOLEAN tipus, igy feltételként BOOLEAN tipusu kifejezés szolgál minden vezérlési szerkezetben. A vezérlési szerkezetek nem zártak, hanem összetett utasitást kell irni a megfelelő helyekre.

IF utasitás

Formája: IF T1 THEN U1 ELSE U2

Az ELSE ág elmaradhat, F1 logikai kifejezés, U1 és U2 összetett

utasitások. Egymásbaágyazott IF szerkezeteknél az ELSE mindig a hozzá legközelebb álló, ELSE nélküli IF...THEN-hez tartozik.

Ciklus utasitások

WHILE utasitások

WHILE Fl DO U

Fl BOOLEAN tipusu kifejezés, U összetett utasitás.
Feltételvizsgálat U végrehajtása előtt. Ha Fl TRUE, akkor U végrehajtódik, különben nem.

REPEAT Ul,U2,...UNTIL Fl

Fl BOOLEAN tipusu, Ul,U2,... utasitások.
Feltételvizsgálat a törzs végrehajtása után.
A törzs ismétlődik, ha Fl értéke false.

FOR utasitás

Formája: FOR Nl:= Kl TO K2 DO U

Nl=változónév, ciklusváltozó, Kl,K2 kifejezések, kezdő és végérték, U összetett utasitás.
TO helyett DOWNTO irható és dekrementálást jelent. A lépésköz mindig a rákövetkező. A ciklusváltozó enumeráció és subrange tipusu is lehet, ilyenkor Kl és K2-nek is olyan tipusunak kell lenni.

CASE utasitás

Formája:
```
CASE Kl OF
Cll,Cl2,...: Ul
    :
Cnl,Cn2,...: Un
END
```

Kl a szelektor kifejezés; a Cll,Cl2,...Cnl,Cn2,... case-cimkék pedig Kl tipusának megfelelő konstansok; Ul,U2,...,Un összetett utasitások. Nincs DEFAULT cimke.

GOTO utasitás

Formája: GOTO cimke

Nem lehet beugrani egy eljárás belsejébe. A cimkéket deklarálni kell az eljárás fejében.

WITH utasitás

Formája: WITH R DO U

R egy rekord neve, U pedig összetett utasitás. U-n belül R mezői

puszta mező neveikkel hivatkozhatók, nem kell minősitett nevet használni.

Eljáráshivás és visszatérés

Az eljáráshivás önálló utasitás.

Formája: N(P1,P2,...)

ahol N az eljárásnév, P1,P2,... az aktuális paraméterek. Négyfajta paraméter osztály van: érték, változó, függvény és eljárás paraméter. Érték paraméter esetén az aktuális paraméter kifejezés lehet, ez az érték szerinti paraméterátadásnak felel meg. Változó paraméter esetén az aktuális paraméternek változónak kell lenni, a hivatkozás szerinti paraméterátadásnak felel meg. Tömbelem paraméternél az index kiszámitása az eljárás hivásakor történik. Visszatérő értéknél mindig változó paramétert kell használni. Az eljárásból való visszatérés az eljárás fizikai végének (END utasitás) elérésekor történik.

Függvényhivás és visszatérés

A függvényhivás formája: N(P1,P2,...)

Ahol N egy függvénynév, P1,P2,... aktuális paraméterek. A paraméterek lehetnek változók, kifejezések, eljárások és függvények és tipusuknak meg kell felelni a formális paraméterek tipusának. A függvényhivás kifejezés része lehet. A függvényből való visszatérés a függvénytörzs fizikai végének elérésekor történik. A törzsön belül kell lenni olyan értékadó utasitásnak, melynek baloldalán a függvénynév áll, ez az érték lesz a visszatérő érték.

4.3.5.3 A PASCAL programszerkezete

A PASCAL programszerkezetére jellemző, hogy a blokkstruktura és a részenkénti fordithatóság lehetősége hiányzik belőle. A program szerkezetét az egymásbaágyazható eljárások és függvények szerkezete adja meg. Összetett utasitás viszont van a nyelvben.

Az összetett utasitás formája: BEGIN U1,U2,...END. Felhasználása a vezérlési strukturákban történik. A változók scope-ját nem befolyásolja.

Az eljárás a programnak egy önálló része, amely felépitésében teljesen hasonlit magára a programra. Az eljárás deklarációk

egymásbaágyazhatók. Az eljárásdeklaráció formája:

PROCEDURE N1 (formális paraméterlista)
cimke deklarációk
konstans definiciók
tipus definiciók
változó deklarációk
eljárás és függvény deklarációk
összetett utasitás.

A fenti deklarációknak és definicióknak szigoruan ebben a sorrendben kell előfordulni. A cimke deklarációkat kivéve mindegyiket tárgyaltuk már. A cimke deklarációk formája:

LABEL C1,C2,...

itt C1,C2,... cimkék.
Az összes cimkét fel kell sorolni, ami előfordul az eljárásban. A formális paraméterlista tartalmazza a formális paraméterek nevét és tipusát, valamint a paraméterosztályt. Az utóbbira vonatkozóan VAR kulcsszó a változó, FUNCTION a függvény és PROCEDURE az eljárás paraméterosztályt jelenti. Értékparaméter esetén a kulcsszó elmarad. Az eljárás nyitott scope, a benne bevezetett nevek lokálisak az eljárásra vonatkozóan. Paraméterként használt eljárásnak vagy függvénynek csak értékparamétere lehet.
Az eljárások lehetnek rekurzivak.
A nyelvben vannak standard eljárások, például a file-kezelést és a dinamikus változók kezelését standard eljárások végzik.

Függvények. A függvény egy olyan programrész, amely értéket számit ki. Az eredmény tipusa skalár, subrange vagy pointer lehet, ez egyben a függvény tipusa is. A függvény deklarációja annyiban tér el az eljárásdeklarációtól, hogy FUNCTION kulcsszó van benne és a formális paraméterlista végén : után az eredmény tipusa áll. A függvények is lehetnek rekurzivak. A nyelvben sok standard függvény van beépitve, aritmetikai függvények, predikátumok, konvertáló függvények és az I/O-t megvalósitó standard függvények.

A program a fejrésztől eltekintve olyan mint az eljárás. A fejrész formája a következő:

PROGRAM N1 (paraméterlista);

Nl a program neve. A paraméterlista csak neveket tartalmaz, olyan egységekre utal, amelyeken keresztül a program a környezetével kommunikál. Ezeket a neveket a programban ugy kell deklarálni, mintha lokális változók lennének.

4.4 Magasszintü SIL nyelvek

Az itt tárgyalt 3 nyelv a középszintü tipusos nyelvek másmás irányu továbbfejlesztései.

A MARY egy ALGOL-68 orientált nyelv, nagy szerepe van a prelude-öknek, a deklarációknak és a makróknak. A nyelv magja majdnem teljesen üres, a legtöbb szintaktikus szerkezet jelentését deklarációval kell megadni. A különböző szintü deklarációkat a standard, implementációs és felhasználó- prelude-ök tartalmazzák. A deklarációk nagyrészét makro-szerüen kell megadni.

A MODULA a PASCAL továbbfejlesztése a párhuzamos programozás irányában. A szekvenciális MODULA a PASCAL egy subsetje, ez kiegészül a zárt scope-ot alkotó MODULE fogalmával és a párhuzamos programozást szolgáló eszközökkel. A MODULA-2 a MODULA továbbfejlesztése, amely a korutint vezeti be párhuzamossági eszközként.

A TARTAN a "legfiatalabb" SIL nyelv. Specifikációja 1978-ban jelent meg. Tulajdonképpen nem SIL nyelvnek, hanem real-time nyelvnek készült, az IRONMEN követelményeit igyekszik kielégiteni.

4.4.1 MARY

A MARY nyelvet Norvégiában a Trondheim-i Egyetem Számitóközpontjában fejlesztették ki Mark RAIN, Per HOLAGER, Reider CONRADI és társaik 1973-ban. A fejlesztők szerint a MARY egy géporientált magasszintü programozási nyelv (MOL), amely szintaktikusan az ALGOL-68-ra és a PASCAL-ra hasonlit.

A MOL nyelveknek igazodni kell azon hardware sajátságaihoz, amelyen futnak. Ez alatt azt értjük, hogy a nyelvnek nem szabad olyan elemeket tartalmazni, amelyek nehezen implementálhatók azon a hardware-en, viszont tartalmaznia kell a hardware adta

lehetőségeket, mégha azok nem szokásosak is. Ehhez az egyik járható ut az, hogy a hardware sajátságait be kell épiteni a nyelvbe. Ekkor viszont a nyelv nem lesz portabilis és nem lesz elég magasszintü. A MARY szerzői egy másik utat választottak. A MARY a gépközelséget két komponensre választja szét. A nyelv magja a minden gépnél közös sajátságokat tartalmazza, a prelude-ök pedig az egyes gépekre jellemző sajátságokat. Igy a nyelv különböző változatai jönnek létre magának a forditóprogramnak a megváltoztatása nélkül.

A nyelvben nagy szerepe van a deklarációknak, a nyelv magja csak azokat a nyelvi elemeket tartalmazza, amelyeket nem lehet deklarálni, például a deklaráció eszközeit és néhány alapelemet. A prelude-ök csupa deklarációt tartalmaznak. A standard prelude tartalmazza azon elemek deklarációit, amelyek közösek a MARY összes implementációja számára. Az implementációs prelude azon elemek deklarációit tartalmazza, amelyek specifikusak az adott hardware-re vonatkozóan, pl. karakterkészlet, vagy amelyek elősegitik a hatékony implementációt pl. speciális operátorok, eljárások stb. A felhasználói prelude egy-egy felhasználói területre orientálja a nyelvet, pl. szövegkezelő, tömbkezelő stb. MARY variánsok. A deklarációk vonatkozhatnak adatra és akcióra.

A MARY kifejezésnyelv, minden szerkezetnek van értéke, még a vezérlési szerkezeteknek is. A program végrehajtása során mindig van egy pillanatnyi érték. Kifejezések kiértékelése közben a pillanatnyi érték a kifejezés már feldolgozott részének értéke. Ez az érték a kifejezés kiértékelésének befejezésekor (pontosvessző elérése) megsemmisül (void lesz), majd uj kifejezés kiértékelése kezdődik.

A MARY variálható nyelv olyan értelemben, hogy a szintaktikus kategóriák értelmezését meg lehet változtatni.

4.4.1.1 A MARY adatszerkezete

A MARY-ban megkülönböztetjük az adat (azaz a futási idő alatt a számológépben tárolt érték) és a pálya (az adathoz való hozzáférési algoritmus) fogalmát. A pályához hozzá tartozik a

mód és három állapotleiró bit: olvashatóság, irhatóság és hivatkozhatóság. Egy adathoz több pálya is vezethet, a pálya legegyszerübb formája az azonositó.

A mód fogalmát a MARY az ALGOL-68-ból vette át. A szokásos tipus fogalom lényeges általánositása. Egy hivatkozási pálya módja definiálja, hogy milyen szintaktikus szerkezet használható ezzel a pályával. Pl. a P(...) programszerkezet csak akkor használható, ha P PROC módu. Ha egy adott környezetben egy adott módu érték nem szerepelhet, akkor a fordító megkisérli a kivánt móduvá kényszeriteni. Pl. A fenti példában ha P REF PROC módu volt, akkor egyszeri indirektséggel PROC móduvá kényszerithető. Tehát a mód tulajdonképpen egy szintaktikus szerkezetet egy pályával hoz kapcsolatba.

A MARY-ben nagy szerepet játszanak a deklarációk. Deklarálni lehet az adatok helyét (cella-, terület- és generátor deklaráció), a hozzáférési pályát (mód deklaráció), a rajtuk végezhető müveleteket (operátor, kényszerités és kontext deklaráció) és van általános makró deklarálási lehetőség. Az adatszerkezetek közül csak a legelemibbek vannak a nyelv magjába belépitve, a felépitményt a prelude-ok deklarációi tartalmazzák.

A MARY-ben a legalapvetőbb adat a halmaz (SET), amely más nyelvek enumeráció tipusának felel meg. Erre épül a prelude-ök deklarációin keresztül a többi hagyományos egyszerü tipus pl. INT, REAL stb. A beépitett összetett és vonatkozó adatszerkezetek a tömb (MULTIPLE), rekord (STRUCUTRE), pointer (REFERENCE) és sor (ROW). Ezekkel kapcsolatban egy alapértelmezés tartozik a [], . és () szintaktikus szerkezetekhez, mégpedig az indexelés, rekord-mező kiválasztás és függvényhivás. Ezeknek az alapértelmezését felüldeklarálhatjuk a MAP-módok alkalmazásával. Rekurziv mód deklaráció is megengedett. Szimultán rekurziv módok az un. modal segitségével deklarálhatók. A mód-minták formális paramétert tartalmazó módleirások, általában összetett módok leirásai.

A halmaz (SET) deklaráció az elemek felsorolását tartalmazza, a felsorolási sorrend egy rendezést definiál.

```
Pl.  SET OPCODE = (ADD,SUB,MUL,DIV)
          OPCODE CODE1:=ADD
```

Itt CODE1 egy OPCODE modu változó, amely az ADD kezdőértéket kapja.

A subrange valamely halmaz egy folytonos tartománya. A subrange módu változó értékkészlete az alaphalmaz folytonos tartományai. Ezek megadása kétféleképpen történhet, vagy

K1 TO K2,

ahol K1 és K2 a kezdő és végértéket megadó konstansok,

vagy K1 for N,

ahol K1 a kezdőértéket megadó kifejezés és N a darabszám.

Példa: SET OPCODE ARITH:=ADD TO SUB

(CODE1 FOR 3) =:ARITH

Itt ARITH egy SET OPCODE (és nem OPCODE) módu cella, amely kezdőértékként az OPCODE halmaz ADD TO SUB tartományát kapja. A második sor kicseréli ARITH értékét arra a tartományra, amely a CODE1 értékével kezdődik és 3 egymást követő elemet tartalmaz, tehát (ADD,SUB,MUL)-ra.

A subset olyan mód, amely konstans subranget definiál. A subset mód értékkészlete egy rögzitett subrange és ennek egy eleme lehet a subset módu változó értéke.

Példa: MODE ADDING=SET (ADD TO SUB)

ADDING CODE2:=SUB

Mint láthatjuk, a szóhasználat erősen eltér a PASCAL szóhasználatától. A megfeleltetés a következő:

értelmezés	MARY	PASCAL
rendezett halmaz	SET	SKALÁR
tartomany	SUBSET	SUBRANGE
hatványhalmaz	SUBRANGE folytonos részhalmazok	(POWER)SET tetszőleges részhalmazok

A hagyományos tipusok deklarációi a prelude-ökben:

SET BOOL = (FALSE,TRUE)

SET FIXED = (...) az egész, illetve

SET CHAR = (...) karakterértékek felsorolása

```
MODE INT  = SET (2**15- TO 2**15-1)
  LONGINT = SET (2**31- TO 2**31-1)
```

A valós mennyiségeket a MARY egészekből alkotott mantissza-exponens pároknak képzeli.

```
MODE REAL = SET (2**15- TO 2**15-1, 127- TO 128)
```

LABEL mód. A MARY nyelvben a cimke egy LABEL módu érték, amely a program valamely pontjára utal. A cimkéhez hozzá tartozik az adott pontban lévő pillanatnyi érték módja és helye is.
Példa: LABEL INT XREG L1;
Ez a deklaráció az L1 cimkét ugy deklarálja, hogy az a program olyan pontjában lesz majd kitéve, ahol a pillanatnyi érték INT módu és az XREG-ben tárolódik. Cimkéket kifejezés belsejében is kitehetünk.
Pl. I+5 L1: *J = :K
A cimke deklarációban a mód és hely specifikátor elhagyható, default értékük VOID, illetve az a hely, ahol a cimke kitételének pillanatában az aktuális érték tárolódik.

Az igy deklarált cimke LABEL módu konstans lesz. Lehet deklarálni LABEL módu változót is, amihez LABEL módu konstans rendelhető értékül. A LABEL mód összetett módok pl. REF LABEL vagy ROW LABEL alapja lehet.

PROC mód. A MARY-ben a szubrutinok is adatként kezelhető szerkezetek. A PROC mód magába foglalja a paraméterek módját és helyét, az eredmény módját és helyét és a generátort, amely a szubrutin tárigényét biztositja.
Pl. PROC RECURS (REAL) INT XREG
egy REAL paraméterü szubrutin, amely tárigényét a RECURS generátortól várja és INT módu eredményét a XREG-ben szolgáltatja.
A PROC módhoz, mint bámely más módhoz változókat deklarálhatunk, amelyekre azonban csak az értékadás és hivás müvelet alkalmazható. Maguk a szubrutinok leirásai PROC módu konstansok, amelyek kezdőértékként rendelhetők PROC módu változókhoz. A rutinleirás megismétli a rutin módjára vonatkozó információkat kiegészitve a formális paraméterek nevének megadásával, ezután a rutin törzse következik, ami BEGIN ... END közötti programrész.
A rutinleirást % jel vezeti be, a törzset :.

```
Pl.  PROC RECURS (REAL) INT XREG P =
         %RECURS (REAL R) INT  XREG :
          BEGIN
           R SIN +5 FIX
          END
```

Itt az első sor a leirás és a cella-deklaráció. A P-nevü PROC módu cella read-only kezdőértéket kap, amit a következő sorok rutinleirása specifikál.

A fenti rutin deklaráció rövidithető

```
    PROC P = %RECURS (REAL R)  INT XREG: BEGIN ... END
```

alakban.

A PROC mód összetett adattipusok alaptipusa lehet.

A tömb (MULTIPLE) azonos módu értékek sorozata, ahol mindenegyes elem megfelel egy index-halmaz egy elemének.

Pl. [BOOL] INT S

Itt S két INT módu értéket tartalmazó tömb, amelyek a TRUE és FALSE index-értékeknek felelnek meg. Az index halmaz rendszerint subset módu.

pl. [SET (1 TO 20)] REAL ARR,

amit röviditeni lehet

```
    [20] REAL ARR
```

alakban.

A tömb mérete forditási idő alatt meghatározható kell hogy legyen, igy a tömbelemek a tárgyprogramban közvetlenül hivatkozhatók. A tömbökhöz az [] szintaktikus szerkezet kapcsolódik. ARR[i] az ARR tömbnek az indexhalmaz i eleméhez tartozó értékét jelöli.

A rekord (STRUCTURE) mezői tetszőleges tipusuak lehetnek. A mezőket névvel azonositjuk.

```
Pl.  MODE COMPLEX = [REAL RE, IM];
     COMPLEX C;
     C.RE+3.5
```

A hivatkozás REKORDNÉV.MEZŐNÉV szintaktikus szerkezettel, azaz minősitett névvel történik.

A pointer (REFERENCE) tartalma egy fizikai hivatkozás egy adatobjektumra.

Pl. INT I;
REF INT RI:=I

Itt RI kezdőértéke az I-hez rendelt rekesz cime. A NIL referen-ce tipusu konstans, azt jelöli, hogy a pointer nem mutat semmi-re. Pointerekre alkalmazható az EMPTY predikátum, értéke TRUE, ha a pointer értéke = NIL, különben FALSE.

Pl. NIL=:RI
IF RI EMPTY THEN ...

Nem szükséges, hogy egy pointer mindig ugyanolyan módu értékre mutasson. Erre az union szerkezet szolgál.

Példa: MODE IRC = REF (INT,COMPLEX, CHAR);
IRC PTR := I
C =: PTR

PTR INT, COMPLEX és CHAR módu értékekre is mutathat, kezdőér-tékként az I INT értékre mutat, majd a COMPLEX módu C-re.

Az union tipusu pointerhez egy mód jelző mező van csatol-va. Az aktuális mód lekérdezésére nyelvi szerkezetek szolgálnak.

IF PTR::-CHAR THEN ...

értéke TRUE, ha PTR aktuális módja érvényes kényszerités lánc-cal CHAR móduvá kényszerithető.

IF PTR=::I THEN ...

Ha PTR aktuális módja lehetővé teszi az egyszerü értékadást, ak-kor az értékadás végrehajtódik és a kifejezés értéke TRUE lesz. Különben értékadás nem történik és a kifejezés értéke FALSE.

A sor (ROW) módu adat lényegében egy tömbleiró elem. Tar-talmaz egy pointert a tömb első elemére, és egy számlálót, a-mely a tömbelemek számát mutatja. Olyan tömbök esetében kell használni, ha a tömbméret csak run-time válik ismertté.

Példa: [132] CHAR BUFFER;
ROW CHAR SUB:=BUFFER [I TO J] ;

Ekkor BUFFER[I] és SUB[1] equivalens hivatkozások.
Egy sor-módu adattal egy tömböt, vagy annak tetszőleges részét leirhatjuk, értéke dinamikusan változhat.

A mód deklarációk során egy módot valamilyen névvel látunk el, és a továbbiakban ezen a néven hivatkozzunk. A MARY-ben két

mód azonos, ha kifejtése azonos. Egyébként nem kötelező a mód elnevezése cella-deklarációkban használhatjuk kifejtett alakját is.

A móddeklaráció jobboldalán állhat beépitett módnév, user definiálta módnév, uj mód, tömb, struktura, REF vagy subset leirása.

Pl. MODE M1 = [10] INT;
MODE KISSZÁMOK = SET (1 TO 100);
MODE REFINT = REF INT;

A rekurziv móddeklaráció megengedi, hogy egy összetett mód leirásában saját magára mutató referencia legyen.

Pl. MODE NODE = [INT VALUE, REF NODE NEXT];

Ezzel lényegében egy lista szerkezetet irunk le. Szimultán vagy közvetett rekurzióról beszélünk, ha két móddeklaráció kölcsönösen tartalmaz a másikra mutató referenciát. Mivel a móddeklarációnak mindig meg kell előznie a felhasználást, ezért a tényleges deklarációk előtt egy fiktiv deklarációt un. modal-t kell alkalmazni.

Pl. MODE FIRST,
SECOND = [REAL VALUE, REF FIRST NEXT],
FIRST = [INT VALUE, REF SECOND NEXT];

A mód minták (MODE-TEMPLATE) formális paraméterrel rendelkező módleirások, ahol a paraméter szintén valamilyen mód.

Pl. MODE LIST (P) = [P VALUE, REF LIST(P) NEXT]

Hivásnál a formális paraméter az aktuálissal helyettesitődik.

Pl. LIST(INT) S:=NIL

hivásnál S módja azonos lesz a fenti NODE móddal.

A MAP-módok alkalmazásával felülirhatjuk bizonyos szintaktikus kategóriák, pl. [], . és () értelmezését. Megtehetjük pl., hogy egy tömbre előirjuk az egydimenziós és kétdimenziós használatot is.

A deklarációk közül a móddeklarációkat már tárgyaltuk. A cella, terület és generátor deklarációkkal 3.2.10-ben foglalkoztunk. Ehhez kiegészitésképpen néhány megjegyzés. A cella deklaráció megadja a cellában tárolódó adat módját, állapotát, kijelöli azt a területet, amelybe kerül és helyet foglal számára. Esetleg kezdeti értékkel is feltölti.

Pl. INT I AREG:=3

jelentése: I nevü INT módu változó az AREG területen a 3 kezdő-

értéket kapja és irható olvasható állapotu lesz. := helyett = operátort használva read-only állapotu változót, azaz konstanst deklarálhatunk.
Példa a terület deklarálásra:

```
GLOBAL AR;
BEGIN INT I AR; ... END;
BEGIN REAL R AR; ... END;
```

Itt az AR területet két blokkban is felhasználtuk, igy az I és R változók ugyanazt a területet foglalják majd el.

A deklarált változók kezdőértéket kaphatnak, ami konstans kifejezés. Tömb és rekord kezdőértéke konstans kifejezések zárójelbe tett listája. Az adatokon végzendő müveletek deklarációjáról, az operátor, kényszerités és kontext deklarációkról szintén 3.2.10-ben volt szó.

A kifejezések kiértékelése szigoruan balról jobbra történik és az operátorok között nincs precedencia.

Monadikus operátorok az operandus után is irhatók. Pl. a következő 3 kifejezés egyenértékü:

```
A+B-
(A+B)-
-(A+B)
```

4.4.1.2 A MARY vezérlési szerkezetei

A MARY viszonylag kevés vezérlési szerkezetet tartalmaz, de ezek nagyon hatékonyak. Vezérlési szerkezeteit nem statementnek, nem is expression-nak, hanem clause-nak, mondatnak hivja. Ezek tulajdonképpen kifejezések, mert értékük van és kifejezések részeként használhatók. A mondat-sorozat (serial clause) szekvenciális feldolgozást jelent, a CASE-mondat esetszétválasztásra szolgál, de a feltételes elágaztatást is magába foglalja. A DO-mondat pedig ciklusszervezésre szolgál.

A mondat-sorozat (serial clause) egymástól pontosvesszővel elválasztott egységek (deklarációk, kifejezések, vezérlési szerkezetek stb.) sorozata. Ezek vegyesen lehetnek, de a deklarációnak meg kell előznie a felhasználást. A mondat-sorozat értéke mindig az utolsó mondat értéke.

A CASE-mondat más nyelvek IF utasitásának és CASE utasitásának szerepét is betölti.

Formája:
```
CASE     U1    IN
  C1:C2:       U2
 CI TO CK:     U3
     :
 OUT           Un
ESAC
```

Itt U1,U2,...Un mondat-sorozatok; C1,C2 értékek, a CI TO CK intervallum minden közbeeső értéket képvisel. OUT CASE összeolvasztható OUSE-vé.

Ha az esetszétválasztás speciálisan logikai értékek szerint történik, akkor a következő helyettesitést végezhetjük:

CASE, IN TRUE:, OUT, OUSE és ESAC

helyett

IF, THEN , ELSE, ELIF és FI

irható.

A DO-mondat ciklusszervezésre szolgál. A ciklus magját DO ... OD kulcsszavak közé kell zárni. Ciklusvezérlő szerkezetek a ciklusmag előtt és után is állhatnak, egyszerre több is. A végrehajtás értelemszerüen a felirási sorrendet követi. A ciklusvezérlő szerkezetek a WHILE, UNTIL és FOR konstrukciók lehetnek. WHILE esetén TRUE értékre, UNTIL esetén FALSE-ra ismétlődik meg a ciklusmag végrehajtása. A FOR szerkezetben egy ciklusváltozót deklarálunk, megadva a kezdő és végértéket, vagy helyette egy subrange-t vagy halmazt, amelynek minden elemét befutja a ciklusváltozó.

Pl. FOR C IN 'A' TO 'Z' DO ... OD;

vagy FOR C IN CHAR DO ... OD;

Ujdonság, hogy a ciklusváltozó tartománya egy sor-módu érték is lehet. Ilyenkor a ciklus változó pointer módu lesz, mégpedig a ROW által leirt tömb elemeinek módjára mutató read-only pointer. A ciklus végrehajtása során a ciklusváltozó sorra a tömb elemeire mutat. Ezzel a szerkezettel a ciklusmagban az indexes változók használata elkerülhető. Mivel az index tulcsordulás vizsgálat a ciklusmagon kivül történik, igy a hatékonyság

lényegesen nő.

```
Példa:    ROW   CHAR R:=SOMETHING;
          FOR  PTR  IN R DO IF PTR='H' THEN
                              ' '=:PTR FI OD;
```

A fenti példa az R stringben minden H karaktert szóközzel helyettesit.

DOWN kulcsszó irásával a lépésirány megforditható.

```
Pl.       FOR PTR DOWN IN R, C DOWN IN CHAR
          DO C =:PTR OD;
```

Az R vektort feltölti az ABC karaktereivel.

A feltétel nélküli vezérlésátadást a standard perlude-ben deklarált GO operátor valósitja meg. A GO operátor kétváltozós, jobboldali operandusa a cimke (kifejezés), amelyre ugrani kivánunk, a baloldali operandusa pedig az aktuális értékként felhasználandó érték (lásd a példát 3.1.1-ben). A LABEL módu változókról 4.4.1.1-ben volt szó.

A szökéskifejezéseket a MARY-ben az ugynevezett range creator segitségével valósithatunk meg.

```
Példa:  DEFINE LEAVE= GO$;
        BEGIN &R:
                 :
              LEAVE R
                 :
        END;
```

&R egyrészt megnevezi a BEGIN és END közötti mondat-sorozatot, másrészt egy konkrét cimkét jelent a sorozat utolsó mondata után, közvetlenül az END előtt. Az erre a cimkére való ráugrás a blokkból való kilépést jelenti. Igy a fenti példában LEAVE R a blokk elhagyását eredményezi. A MARY forditó minden rutintörzset illetve minden ciklusmagot automatikusan ellát a ROUTINE illetve LOOP range-ekkel. Igy LEAVE ROUTINE és LEAVE LOOP kifejezéseket irhatunk, ami a megfelelő egységből való kiugrást eredményez.

Eljáráshivás és visszatérés

Az eljáráshivás alakja:

E(P1,P2,...Pn)

Itt E egy PROC módu értéke eredményező kifejezés, P1,P2,... pedig az aktuális paramétereket megadó kifejezések.

Példa: ROW PROC (INT) REAL ROWER:=NIL

deklarál egy eljárástömbre mutató ROW módu elemet, ahol az eljárások mind INT paraméterüek és REAL eredményt szolgáltatnak.

A ROWER[I+5](K+2)

kifejezés hivja az I+5-ödik eljárást K+2 aktuális paraméterrel.

Az eljárások paraméterátadása érték szerint történik. A referencia szerinti paraméterátadás a referenciák érték szerinti átadásával, a név szerinti paraméterátadás pedig az eljárások érték szerinti átadásával valósitható meg.

4.4.1.3 A MARY programszerkezete

A MARY tipikusan kifejezésnyelv, minden kifejezés a nyelvben, még a vezérlési szerkezetek is, mindennek van értéke. Más nyelvek összetett utasitásának a mondatsorozat (serial clause) felel meg. A mondatsorozat elemi mondatok ;-vel elválasztott sorozata. Egy elemi mondhat lehet kifejezés, deklaráció vagy egy vezérlési szerkezet. Egy mondatsorozatban deklarált változó scope-ja a deklaráció helyétől a mondatsorozat végéig tart. A mondatsorozat lehet a BEGIN ... END kulcsszavak közé téve, ekkor zárt mondatsorozatnak nevezzük. A mondatsorozatok egymásba ágyazhatók. A zárt mondatsorozat nyitott scope, tehát egy mondatsorozaton belül deklaráció nélkül használhatjuk globális változóként. Blokk nincs a nyelvben, a blokk fogalomnak a zárt mondatsorozat felel meg. Azonban zárt mondatsorozat csak speciális helyen állhat, ilyen pl. a főprogram vagy az eljárások törzse.

Eljárások. Mint a 4.4.1.1-ben már láttuk, az eljárások nevei PROC módu változóknak tekintődnek a nyelvben. Az eljárás törzse zárt mondatsorozat, tehát nyitott scope-ként viselkedik. Az eljárások lehetnek rekurzivak.

4.4.2 MODULA-2

A MODULA-2 nyelvet N.Wirth hozta létre az Eidgenössische Technische Hochschule, Zürichben. A nyelvet elsősorban kisgépeken történő folyamatvezérlési rendszerek készitésére fejlesztették ki. A hatékony implementálhatóság és a hardware-sajátságokhoz való hozzáférés a nyelv specifikálás egyik döntő szempontja volt. A nyelv eredeti változata, a MODULA 1977-re készült el. A MODULA a PASCAL nyelv erősen leegyszerüsitett vázára épül. Pl. nincs benne file, változó rekord, subrange, real szám stb. Ez a váz kiegészül a MODULE fogalommal és a multiprogramozás eszközeivel, azaz a PROCESS, SIGNAL és INTERFACE MODULE fogalmával. A MODULA-2 a MODULA továbbfejlesztett változata. Az eltérés egyrészt az, hogy a PASCAL-ból elhagyott fogalmak egy részét beépiti a nyelvbe, másrészt a MODULA-ban használt párhuzamos programozási eszközök helyett alacsonyabb szintü eszközt, a korutin fogalmát vezeti be. A továbbiakban a MODULA-2 nyelvet ismertetjük.

4.4.2.1 A MODULA-2 adatszerkezete

A MODULA-2-ben a következő adattipusok vannak:

Egyszerü önálló beépitett: INTEGER, CARDINAL, BOOLEAN, CHAR. A CARDINAL tipus a nem-negativ egészeket jelöli.

Egyszerü önálló definiált: enumeráció, procedure.

Egyszerü vonatkozó: subrange, set, pointer.

Összetett önálló: array, rekord.

Összetett vonatkozó: BITSET = SET OF 0 .. 15

Ezek közül az enumeráció, subrange, pointer, array és rekord teljesen megegyezik a PASCAL-ban használt fogalmakkal. Változó rekord is van. A set abban különbözik a PASCAL-ban használttól, hogy alaptipusának értékkészlete nem tartalmazhat 16 értéknél többet. Procedure tipus a PASCAL-ban nem szerepel, sőt a MODULA-ban sem. A tipus maga tartalmazza a lokális paraméterek és az eredmény tipusát. A procedure tipusu változó értéke egy procedure leirás lehet. A formális paraméterek tipusainak rendre meg kell egyezni a tipus-leirásban és a procedure leirásban, de a procedure leirásban a formális paraméterek neve is szerepel.

A nyelvben használható konstansok az egész szám, karakter, string és setkonstans. A setkonstans a set alaptipusának neve után zárójelben megadja a konstans értékéhez tartozó elemeket, vagy intervallumokat.

```
Pl.  TYPE  NAPOK = (HÉTFŐ, KEDD, SZERDA, CSÜT, PÉNT, SZOMB, VAS)
     T  = SET OF NAPOK
     VAR N:T
     N := NAPOK {HÉTFŐ, SZERDA .. PÉNT}
```

A változó deklaráció változó nevet és tipust rendel össze. Memória osztály és kezdőértékadás nincs a nyelvben. A konstans, tipus és változódeklaráció kulcsszava rendre CONST, TYPE és VAR. Ezeken kivül eljárást és modult lehet még deklarálni.

A müveletek tipusfüggőek és 4 precedencia osztályba tartoznak a NOT operátor, a multiplying, adding és reláció operátorok osztályába. Több különböző operátor van ugyanazzal a szimbólummal jelölve, ilyenkor az operandusok tipusa dönti el, hogy melyik müveletet kell alkalmazni. Aritmetikai, logikai, bitenkénti logikai (set müvelet) és reláció müveletek vannak a nyelvben.

Az összetett tipusok csak elemeiken keresztül hivatkozhatók. A subrange tipuson ugyanazok a müveletek végezhetők, mint az alaptipuson. A karakter, enumeráció és pointer tipusra csak a relációk értelmezhetők. A procedurát csak hivni lehet.

4.4.2.2 A MODULA-2 vezérlési strukturái

A vezérlési szerkezetek zártak, azaz mindegyik az END záró kulcsszóval végződik, a feltétel logikai kifejezés.

IF utasitás

```
Formája: IF   F1 THEN U1
         ELSIF F2 THEN U2
           :
         ELSE  Un
         END
```

Itt U1,...,Un utasitás sorozatok, F1,F2,... feltételek.

Ciklus utasitások

WHILE F1 DO U END

Itt F1 feltétel, U utasitás sorozat. Feltételvizsgálat U végrehajtása előtt, a végrehajtás F1=TRUE-ra ismétlődik.

REPEAT U UNTIL F2

U utasitássorozat, F2 feltétel. Feltételvizsgálat U végrehajtása után, ismétlés F2=FALSE értékre.

FOR N1:=K1 TO K2 BY K3 DO U END

Itt N1 név, K1, K2 kifejezések, K3 konstans, U utasitássorozat. BY K3 elmaradhat, default értéke=1. K1 és K2 a ciklusváltozó kezdő és végértéke, K3 a lépésköz.

LOOP U END

U utasitássorozat. Látszólag végtelen ciklus, a ciklusból való kiugrást az EXIT utasitás teszi lehetővé.

Case-utasitás

Formája:
```
CASE   K1   OF
C11,C12,...: U1
      :
Cn1,Cn2,...: Un
ELSE  Un+1
END
```

Itt a K1 szelektor kifejezés tipusa nem lehet összetett tipus. C11,C12,...Cn1,Cn2,... case-cimkék, ezek tipusának meg kell egyezni a szelektor kifejezés tipusával. U1,...,Un+1 utasitás sorozatok. Az ELSE ág elmaradhat.

Feltétel nélküli vezérlésátadás nincs a nyelvben.

WITH utasitás

Formája: WITH R DO U END

R egy rekordnév, U egy utasitássorozat. U-ban R mezői egyszerüen a mezőneveikkel hivatkozhatók, nem kell minősitett nevet használni.

Az EXIT szökéskifejezés a LOOP utasitásból való kiugrást teszi lehetővé.

Eljáráshivás és visszatérés

Formája: N(P1,P2,...)

N PROC tipusu kifejezés, P1,P2,... aktuális paraméterek. A paraméterek lehetnek érték vagy változó paraméterek. Az érték paraméternek kifejezés, a változó paraméternek változó felelhet meg az aktuális paraméterek között. Az eljárás törzsből való visszatérést a RETURN E utasitás valósitja meg. Az E kifejezés értéke a visszatérő érték, ha elmarad, akkor nincs visszatérő érték. Az eljárástörzs fizikai végének elérése is visszatérést eredményez.

Processz

A MODULA-2 alapvetően a hagyományos egyprocesszoros gépeken való implementálásra készült. Tulajdonképpen csak kvázi-konkurens folyamatokat enged meg, és a perifériás készülékekkel való párhuzamosságot. A processz szót tulajdonképpen "korutin" értelemben használja, egyszerre mindig csak egy processz kaphatja meg a vezérlést. A processzek kezelésére két eszköz van, a létesitás és a vezérlésátadás. Egy processz létesitése a

NWEPROCESS(P,A,n,P1)

utasitással történik.

Itt P egy fő-szintü paraméternélküli eljárást jelöl meg, A a processz munkaterületének kezdőcime, n pedig a hossza, P1 egy processz tipusu változó. Az utasitás hatására a kivánt processz létrejön, hozzárendelődik P1-hez, de nem aktiválódik. A vezérlést egy TRANSFER utasitás hatására kapja majd meg.

A TRANSFER(P1,P2)

utasitás az aktiv processzt felfüggeszti, hozzárendeli P1-hez és a vezérlést a P2-höz rendelt processznek adja.

4.4.2.3 A MODULA-2 programszerkezete

A MODULA nyelv egyik legjelentősebb ujitása a zárt scope-ot alkotó MODULE fogalom bevezetése. A nyelv utasitás-orientált. Összetett utasitás nincs, mert a vezérlési szerkezetek záró kulcsszóval végződnek. A blokk is csak mint eljárástörzs vagy modultörzs jelenik meg, önálló blokk nincs.

Eljárások. Az eljárás deklaráció egy fejet és egy törzset tartalmaz. A fej tartalmazza az eljárás nevét és a formális pa-

ramétereket, a törzs egy blokk, amely BEGIN ... END között deklarációkat és utasitásokat tartalmaz, az END után az eljárás neve megismétlődik. Kétfajta eljárás van. A függvény-eljárásnak van visszatérő értéke és hivása kifejezés részeként szerepelhet. A függvény-eljárás törzsének tartalmazni kell egy RETURN utasitást. A valódi eljárás (proper procedure) hivása önálló utasitás, visszatérő érték nincs. Az eljárástörzs nyitott scope, azaz a benne deklarált minden konstans, változó, tipus, modul és eljárás lokális a törzsön belül. A kivül deklarált változók pedig globálisak, azaz a törzsön belül is hivatkozhatók. Az eljárások rekurzivak. A paraméterek tipusa bármi lehet, tömb és procedure tipus is. Eljárás paraméter esetén az aktuális paraméternek vagy egy főszinten deklarált eljárásnak kell lenni vagy egy PROC tipus változónak és nem lehet standard eljárás.

A MODULE deklarációk és utasitások sorozata a MODULE...END kulcsszavak közé zárva. A MODULE zárt scope-ot alkot (lásd 3.1.5). A modul-fej tartalmazza a modul nevét valamint az import és export listát. Az IMPORT lista tartalmazza az összes kivül deklarált, belül használt nevet, az EXPORT lista pedig a belül deklarált, kivül használt neveket. A modul törzs egy blokk, ami után megismétlődik a modulnév. QUALIFIED export esetén a felsorolt azonositókra minősitett névvel (modulnév.változónév) lehet csak hivatkozni. IMPORT lista esetén a FROM modulnév prefix feloldja qualified export követelményeit.

4.4.3 TARTAN

A TARTAN nyelv specifikációja W.A.Wulf munkája. Tervezésénél két alapvető szempont volt, hogy elégitse ki az Ironman követelményeit és hogy egyszerü legyen a nyelv. Wulf ismerte az Ironman követelményrendszerhez készült nyelvek tervezeteit, de mindegyiket tul bonyolultnak találta. A nyelvet egy kisérletnek szánta arra, hogy egyszerü nyelv is kielégitheti az Ironman követelményeit. A tervezésre mindössze 2 hónap állt rendelkezésére (ápr. 1-től május 30-ig), a nyelv néhány részlete nincs is kidolgozva. A nyelv felhasználja a rendszerprogramozási nyelvek terén elért összes eredményt, pl. a modul és processz fogalmát a

MODULA-ból, a változó rekord-ot és a dinamikus változót a PASCAL-ból. Vezérlési strukturái a strukturált programozásnak leginkább megfelelő kicsiszolódott strukturák. Bevezet néhány uj fogalmat is, pl. bevezeti a konstruktort az összetett és dinamikus tipusok számára és az exception fogalmát a hiba-helyzetek kezelésére. Tisztázza a változók scope-jával kapcsolatos fogalmakat, megkülönbözteti a nyitott és zárt scope fogalmát.

4.4.3.1 A TARTAN adatszerkezete

A TARTAN tipusos nyelv. Minden objektumnak fix tipusa van, amely a forditás során határozódik meg. A TARTAN megkülönbözteti a tipus és a tipusnév fogalmát. Egy tipus legfontosabb jellemzői: a literáljainak formája, paraméterei (attributumok), a rajta müködő infix operátorok, speciális operátorok és standard rutinok; ezek a beépitett tipusokra pontosan rögzitve vannak a nyelv specifikációjában.

Pl. Float tipus:

Literal: számjegysorozat tizedesponttal

Attributumok: Min, Max, Radix, Precision, MinExp, MaxExp.

Infix operátorok: aritmetikai operátorok és relációk

standard rutinok: kerekités és csonkitás

A felhasználó definiálta tipusokra ezeket a felhasználó irja elő. A tipusnév tartalmazza a tipusellenőrzéshez szükséges összes információt. Pl. az ARRAY (A.. B) OF D tipus tipusneve "ARRAY[I,D]", ahol I az index range tipusa, D pedig az alapelemek tipusa. Egy tömb indexeinek határai és általában egy tipus attributumai nem szerepelnek a tipus ellenőrzésben

A TARTAN-ban szereplő beépitett tipusok: FIXED, FLOAT, BOOLEAN, CHAR, LATCH, ENUM, range, ARRAY, RECORD, VARIANT, SET, STRING, FILE, ACTIVATION, ACTNAME és DYNAMIC tipus. Ezek közül csak azokkal foglalkozunk, amelyek vagy uj fogalmak, vagy valamiben eltérnek a hagyományostól.

A LATCH (retesz, zár) a párhuzamos programozás egyik eszköze, a kölcsönös kizárást szolgálja. Állapotai OPEN és CLOSED. A rajta operáló standard rutinok: LOCK, IFLOCK, UNLOCK és az ACTIVATION rutinok.

A range K1.. K2 alaku, ahol K1 és K2 konstansok.

A VARIANT a PASCAL nyelv változó rekordjának egyszerüsitett változata. Egyetlen tag-field-je van, amely a rekord legelején helyezkedik el.

Formája:

VARIANT N:T[ON K1 → T1 ON K2 → T2 ... ON OTHERS → Tn]

Itt N változónév, T tipusnév, K1,K2,... T tipusu konstansok, T1,T2,...,Tn tipusnevek. Az OTHERS kulcsszónak leghátul kell szerepelnie és az összes nem szereplő értéket összefoglalja.

Példa:

```
VARIANT A: NAPOK [ON HETFŐ → INT  ON  KEDD →
     REAL  ON  OTHERS → CHAR]
```

A SET egy boolean vektor, amelyen logikai müveletek végezhetők és indexelni lehet. Paramétere a maximális hossz.

A STRING egy karakter vektor, amelyen müveletek végezhetők. Paramétere a string hossza. A rajta végezhető speciális müveletek a karakter kiemelés (subscript), alstring kiemelés (substring), és egymásutánfüzés (catenation).

A FILE egy minimális I/O tevékenység megszervezésére szolgál.

Az ACTIVATION és ACTNAME tipusu változók processzek kezelésére szolgálnak. A processz olyan rutin, amely felkészült a párhuzamos futtatás lehetőségére, azaz egyszerre több példánya is létezhet. Egy ACTIVATION a processz egy példánya, az ACTNAME pedig egy processz példányra mutató pointer. Mig az ACTIVATION egy rögzitett processzhez kötött, addig az ACTNAME tetszőleges processz példányára mutathat. Az ezeken a változókon müködő standard rutinok és speciális operátor (CREATE) leirását 4.4.3.2-ben találjuk meg.

A DINAMIC tipusu változó tulajdonképpen a PASCAL pointer tipusu változójának utóda, azaz egy tipusra mutató pointer a hivatkozott objektummal együtt. A változó deklarációjában rögziteni kell, hogy mely tipusra mutat, de csak a pointer jön létre és NIL kezdőértéket kap. Értéket (kezdőértéket is) ugynevezett konstruktoron keresztül kaphat, ami konstansokból egy adott adattipus szerkezetének megfelelően felépitett adatsokaság.

A konstruktort egy tipus jelző prefix előzheti meg. A dinamikus tipusu konstruktor feldolgozásának eredménye egy pointer egy uj objektumra, amely objektum a konstruktor adatait tartalmazza és tipusa megegyezik a konstruktor tipusával. Mindenegyes konstruktor egy uj példányt hoz létre a tipusból, ami addig marad életben, amig van rá hivatkozás. Mivel a dinamikus tipus paraméterei csak akkor derülnek ki, amikor a konstruktort használjuk, ezért van rá lehetőség, hogy egy dinamikus változóhoz különböző időpontokban különböző paraméterekkel rendelkező objektumokat rendeljünk hozzá. A dinamikus rekord mezőjére közvetlenül <név>.<mezőnév> szerkezettel hivatkozhatunk. Bármilyen tipusu dinamikus változó esetén <név>.ALL kifejezés hivatkozik a változóhoz tartozó objektum teljes egészére, azaz minden komponensre együtt.

Ezeken a beépitett tipusokon kivül a felhasználó uj tipusokat is definiálhat.

A nyelvben a következő konstansok használhatók: számok tizedesponttal vagy anélkül, TRUE, FALSE (boolean értékek), NIL (dinamikus változó kezdőértéke), OPEN, CLOSED (latch értékek), MINT (processz kezdőértéke), EMPTY (set érték), enumeráció elemei, definiált konstans nevek, konstans kifejezések, tipusnévvel prefixált konstans és konstruktor.

A tipusnévvel prefixált konstans esetében a konstans tipusa a prefixként megadott tipus lesz. Olyan esetben célszerü használni, amikor a konstans lexikális formájából nem derül ki, hogy milyen tipusunak szánjuk.

Pl. REAL'1

Egyébként tipusnévvel és modulnévvel változókat is lehet prefixálni, ilymódon minősitett nevet kapunk. Ezáltal megengedhető, hogy ugyanazt a nevet különböző tipusu változó jelölésére használjuk.

```
Pl.  TYPE SZINEK := ENUM (PIROS, KÉK, ZÖLD, FEKETE)
     TYPE KÖZLLÁMPA  := ENUM (PIROS,SÁRGA, ZÖLD)
     VAR  V1 : SZINEK
     VAR  V2 : KÖZLLÁMPA
     V1 := SZINEK'PIROS; V2:= KÖZLLAMPA'PIROS;
```

A konstruktor konstansként szolgál összetett tipusok és dinamikus tipusok számára. Lényegében konstansokból álló adatstruktura, amelynek szerkezete megfelel egy adott tipus adatszerkezetének. A dinamikus tipusnál való használatát már láttuk. Használhatjuk még rekord, tömb és variant kezdőértékadására, illetve értékadására.

Formája: (E1,E2,...)

vagy (N1→E1, N2→E2,..., OTHERS→En)

itt E1,E2,...,En kifejezések, N1,N2,... pedig egy range elemei. OTHERS jelenti az elő nem fordult értékek együttesét.

Definició és deklaráció. A TARTAN megkülönbözteti a definiciót és deklarációt. A definició leköt egy nevet egy modulhoz, rutinhoz (eljárás vagy függvény), processzhez, tipushoz vagy exception-hoz, feldolgozása a forditás során történik. A deklaráció leköt egy nevet egy objektumhoz (változó vagy érték), feldolgozása futási időben történik, rendszerint helyfoglalással.

A tipus definicióval a felhasználó uj tipust vezethet be a programjába. A tipus definició bevezeti a tipusnevet és definiálja a tipus reprezentációját. Az uj tipuson végezhető müveleteket rutin definiciókkal vezethetjük be és az alaptipuson végzett müveletekre vezetjük vissza. Erre a REP prefix szolgál.

Példa:
```
TYPE Mark = INT;
FUNC "+" (a,b:Mark) c:Mark;
BEGIN REP'C:=REP'a + REP'b END;
```

A változó deklaráció formája:

<binding> L:T:=K

ahol L változónév lista, T tipus, K konstans kezdőérték és

<binding> :=empty|VAR|CONST|MANIFEST|RESULT

T és K közül az egyik elmaradhat.

VAR = változó, CONST = read only változó, MANIFEST = forditási idő alatt meghatározódó konstans, RESULT = rutin output változója.

4.4.3.2 A TARTAN vezérlési strukturái

A TARTAN vezérlési strukturáit záró kulcsszó zárja, igy utasitás-sorozatok is szerepelhetnek bennük. IF, WHILE, FOR, CASE és GOTO utasitás szerepel a nyelvben, ezenkivül van rutinhivás, processz vezérlés és hiba lekezelés.

A feltételes utasitás

```
IF F1 THEN U1
ELIF F2 THEN U2
  :
ELIF Fn THEN Un
ELSE Un+1
FI
```

Itt F1,F2,...Fn boolean kifejezések, U1,U2,...,Un,Un+1 utasitássorozatok. Az ELIF részek és az ELSE rész is elmaradhat.

Ciklus utasitások

```
WHILE F DO U OD,
```

Itt F feltétel, U utasitássorozat.
Feltételvizsgálat a törzs végrehajtása előtt, ismétlés F=TRUE esetén.

```
FOR N IN R DO U OD
```

N = a ciklus változó neve, R egy range, U utasitássorozat. A ciklus az R range minden elemére végrehajtódik.

Esetszétválasztás

Formája:

```
CASE EO
ON E11,E12,...      → U1
ON E21,E22,...      → U2
     :
ON En1,En2,...      → Un
ON OTHERS           → Un+1
ESAC
```

EO-nak FIXED vagy ENUM tipusunak kell lenni. Az Eij-k pedig MANIFEST konstansok, mégpedig EO tipusának elemei, esetleg range-i. OTHERS az összes nem szereplő esetet összefoglalja U1,U2,...,Un+1 utasitássorozatok, amelyek lezárását az ON ismétlődése adja.

Feltétel nélküli vezérlésátadás

Formája: GOTO L, ahol L cimke.
Rutin vagy modul törzséből nem lehet kiugrani, beugrani pedig sem blokkba sem rutinba nem szabad.
Szökéskifejezések a TARTAN-ban nincsenek.

Rutinhivás (eljárás és függvény)

A rutinhivás megadja a hivandó rutin nevét és az aktuális paramétereket. Az eljárás hivás önálló utasitás, mig a függvény hivás kifejezés, amelynek értéke van. Az aktuális paraméterek tipusának ugy kell megfelelni a formális paraméterek tipusának, hogy tipusnevüknek egyezni kell, de a tipus paramétereinek nem kell feltétlenül megegyezni. Az aktuális paraméterek lekötésének a következőképpen kell viszonyulni a formális paraméterek lekötéséhez.

formális paraméter	aktuális paraméter
VAR	változónév
CONST	kifejezés
MANIFEST	manifest kifejezés
RESULT	változónév

A CONST és MANIFEST paraméter lényegében az érték szerinti paraméterátadásnak felel meg, a VAR és RESULT paraméter a hivatkozás szerinti paraméterátadást valósitja meg. A RESULT paraméter számára létrejön egy ugyanolyan tipusu lokális változó, amelynek értéke a rutinból való kilépéskor bemásolódik az aktuális paraméterbe. Minden aktuális paraméter belépéskor értékelődik ki (pl. indexes változó).

Processz-vezérlés

A processzek kezelésére szolgáló ACTIVATION és ACTNAME tipusu változókról már volt szó a TARTAN adatszerkezetének leirásánál.

Egy processznek több példánya lehet, amelyek mindegyike a MINT, SUSPEND, ACTIVE és DEAD állapotok valamelyikében van.

Egy processz definiálásakor megadjuk a formális paramétereket és a program törzset; ebből csak egy minta készül, konkrét példány nem jön létre. Az ACTIVATION deklaráció létrehoz egy névvel ellátott konkrét példányt és azt MINT (vadonatuj) állapotba teszi. Ezt a példányt CREATE NÉV (paraméterek) utasitással lehet életre kelteni. Itt NÉV egy példány neve, aminek tipusa egyértelmüen megszabja, hogy mely processzről van szó. A paraméterek pedig a processz formális paramétereinek megfelelő aktuális paraméterek. A CREATE utasitás hatására megtörténik a paraméterátadás és a processz példány SUSPEND állapotba kerül.

Példa:
```
PROCESS P (VAR A:REAL)
  BEGIN ... END
VAR X : ACTIVATION OF P
VAR C : REAL
CREATE X(C)
```

A továbbiakban a processz példány állapota a SUSPEND és ACTIVE állapotok között váltakozik egészen haláláig, amikor is DEAD állapotba kerül.

Mind az ACTIVATION, mind az ACTNAME objektumokon ugyanazok a standard rutinok alkalmazhatók. Ezek a következők:

Állapotváltozást okozó rutinok:

Activate (A), Suspend (A), Terminate (A),
LockAndActivate (A,L), LockAndSuspend (A,L),
UnlockAndActivate (A,L), UnlockAndSuspend (A,L)

Állapot lekérdező rutinok:

IsMint (A), IsAct (A), IsSusp (A), IsTerm (A)

Egyéb rutinok:

NameOf (A), Notify (A), Priority (A), SetPriority (A),
Time (A).

Itt A Activation és L latch.

Az Activation és Actname tipusu változókon az átutaláson kivül más müvelet nem végezhető, az is csak MINT állapotban.

Az exception valamilyen speciális körülmény, általában hiba miatt bekövetkező rendkivüli állapot. Minden exceptionnak neve van, amit deklarálni lehet. Egy exception-név hatásköre az a blokk, amelyben deklarálták. Egy blokk belső blokkjában el lehet nyomni az exceptiont, erre a DISABLE deklaráció szolgál.

Példa:
```
EXCEPTION TooBig, TooSmall, Late, Singular
DISABLE   TooBig, TooSmall
```

Standard exception is van, pl. Assertion.

Az exception-nel kapcsolatos utasitások a SIGNAL, RESIGNAL és ASSERT. Az exception-ök feldolgozását az egyes utasitásokhoz rendelt handler egységek (handler clause) végzik. Ha egy exception fellépett, a vezérlés átadódik a legközelebbi dinamikusan beágyazott handler egységre és ez fut le az utasitás második része helyett. Ha ez a handler nem a pillanatnyilag végrehajtás

alatt álló blokkban van, minden közbeeső blokk terminálódik. Ha a handler nincs a pillanatnyilag végrehajtás alatt álló rutinban, akkor a rutin terminál és az exception ujra fellép a rutinhivás pontján. Ha nincs handler az exception-név scope-jában, akkor egy default handler fogja terminálni a blokkot.

Egy handler-egység több exceptiont is lekezelhet, mindegyikhez egy

ON exceptionnév → utasitások

alaku rész tartozik. A maradék eseteket az OTHERS cimke gyüjti össze.

Az ASSERT utasitás equivalens a következővel:

IF <expr> THEN SIGNAL Assertion FI;

Példák:

```
SIGNAL TooBig
ASSERT  X<O
:
READ (file,X) {ON EOF → GOTO Exit}
X:=X+1  {ON Overflow → X:=O}
```

4.4.3.3 A TARTAN programszerkezete

A TARTAN tisztázza a programszerkezettel és a változók scope-jával kapcsolatos fogalmakat megteremtve a nyitott és zárt scope fogalmát. A TARTAN-ban a következő program egységek fordulnak elő: utasitás, blokk, modul, rutin (eljárás és függvény) és processz.

A TARTAN utasitás orientált nyelv. Az utasitások lehetnek definiciók, deklarációk és végrehajtható utasitások. A végrehajtható utasitások az eljáráshivás, az értékadó utasitás, a vezérlési szerkezetek, a blokk és az üres utasitás.

Összetett utasitás nincs a nyelvben, a vezérlési szerkezetek záró kulcsszóval vannak ellátva.

A blokk nyitott scope. Formája:

BEGIN D1,D2,...,U1,U2,...END

ahol D1,D2,... definiciók és deklarációk, U1,U2,... utasitások. A blokkba csak a blokkfejen keresztül lehet belépni. A definicióknak és deklarációknak mindig elől kell állni. Először mindig ezek feldolgozása történik meg a felirás sorrendjében, bele-

értve a rutin, processz és modul definiókat is. Utána a végrehajtható utasitások feldolgozása következik. Végül a blokk vagy kilép (it is exited) vagy megszakad (it is terminated). Ha kilép akkor a vezérlés megvárja, amig minden, a blokkban deklarált processz-példány DEAD vagy MINT állapotba jut, azután a blokkban deklarált minden nemdinamikus változó helye felszabadul. A blokk megszakad, ha egy exception lép fel benne vagy egy beágyazó processz terminál, ilyenkor a blokkban deklarált minden processz-példány azonnal terminálódik és a blokk kilép.

A modul zárt scope-ot alkot.
Formája: MODULE N1 B1 , ahol N1 név, B1 blokk.
A modul lényeges része az export-, import-lista, amely az exportált illetve importált neveket tartalmazza. Ezek lehetnek változó-, modul-, rutin-, tipus- és exception-nevek . Nincs megengedve a változónevek ujradefiniálása egy scope-on belül. Ugyanakkor lehet importálni ugyanazon nevet különböző scope-okból különböző jelentéssel. Az ilyen neveket prefixálni kell a definiáló modul nevével. Ha kétértelmüség nem léphet fel, akkor a prefix elmaradhat.

A modul deklarációjának feldolgozásakor feldolgozódnak a modul-törzs deklarációi lexikális sorrendben és végrehajtódnak a modul utasitásai. A modulban definiált rutinok kivülről akkor hivhatók, ha exportáljuk őket.

A rutin egy zárt scope, amelynek törzse egy blokk. Lehet eljárás, függvény vagy processz. A deklaráció formája:

```
PROC N (P1,P2,...); B
FUNC N1 (P1,P2,...) N2:T;B
PROCESS N(P1,P2,...) ; B
```

Itt N, N1 és N2 nevek, P1,P2,... formális paraméterek, T tipus és B blokk. A formális paraméterek megadási módja:

<prefix><változó név> : <tipusnév>,

ahol

<prefix>:: = VAR|CONST|MANIFEST|RESULT

Az eljárás módosithatja akutális paramétereit és hivása önálló utasitás. A függvénynek visszatérő értéke van és mellékhatása nincs, a processz felkészült a párhuzamos futtatás lehetőségére. A rutinnevek felülirhatók olyan értelemben, hogy ugyan-

az a rutinnév szolgálhat több különböző paraméterü rutin neveként. Ilyenkor híváskor az aktuális paraméterek tipusa alapján dől el, hogy melyik rutint hivják.

A processz scope szempontjából rutinként viselkedik. A párhuzamos futtatás eszközeivel 4.4.3.2-ben már foglalkoztunk.

4.4.3.4 Generic definició

```
Példa:  GENERIC MODULE RingDef [K:Int];
        BEGIN
        EXPORTS Ring, Next;
        TYPE  Ring=FIXED  (1,0,0,K-1);
        FUNC  Next(R:Ring) N:FIXED  (1,0,0,K-1);
            BEGIN N:= mod  (R+1,K); END
        END
        MODULE R5 Is  RingDef [5];
        MODULE R9 Is  RingDef [9];
```

A generic definició tulajdonképpen makró definiciónak felel meg. GENERIC kulcsszóval kezdődik és a definiált név után szögletes zárójelben tartalmazza a generic paramétereket. A generic definició hivásának eredménye a definició törzsének bemásolása a hivás helyére, mégpedig a formális paraméterek helyére az aktuális paramétereket helyettesitve. Aktuális paraméter bármilyen, már definiált név lehet.

IRODALOMJEGYZÉK

1. Gyárfás András: A PL360 programozási nyelv. MTA SZTAKI Tanulmányok 68/1977.
2. PLM. MTA SZTAKI IDOS MANUALS 1978. Belső használatra
3. Richards, M.: BCPL: A tool for complier writing and systems programming. AFIPS 34 (1969) Proceeding of the Spring Joint Computer Conference
4. Revised BCPL Manual, April 28, 1975.
5. A BCPL INFELOR változatának kézikönyve. INFELOR Rendszertechnikai Vállalat Programozási Rendszerek Főosztálya. 1974 (Inf. 1416/1974)
6. Wulf, W.A. and others: BLISS: A Language for Systems Programming. Communication of the ACM 1971 december Vol. 14 No. 12.
7. XXPL (magasszintü programozási nyelv a rendszer és alapsoftware technológia fejlesztésére). Távközlési Kutató Intézet, Intézeti Tanulmány 1975.
8. Nagy Antal: XPL: alapsoftware irására alkalmas programnyelv. Információ Elektronika 1974/3.
9. McLeod, R.: Edinburgh IMP Language Manual, Edinburgh Regional Computing Centre 1974.
10. Stephens, P.D.: A Syntax and Semantic definition of the IMP Language. Edinburgh Regional Computing Centre 1974.
11. Ritchie D.M.: C Reference Manual. Bell Telephone Laboratories. Murray-Hill, New Yersey, 1974.
12. Farkas Ernő: GESAL. MTA SZTAKI 1980. Belső használatra.
13. Hermann Gábor: Bevezetés a GÉZA nyelv használatába. MTA SZTAKI 1980. Belső használatra.
14. Gerhardt Géza: Design and Implementation of a High Level System Programming Language. Second Hungarian Computer Science Conference. Budapest, 1977.

15. Wirth, N.: The Programming Language PASCAL. ACTA Informatica. 1. 35-63. (1971)

16. Wirth, N.: The Design of a PASCAL Compiler. Software Practice and Experience, 1. 309-333. (1971)

17. Haberman A.N.: Critical Comment on the Programming Language PASCAL. Acta Informatica 3, 47-57. (1973).

18. Hoare, C.A.R. and N. Wirth: An Axiomatic Definition of the Programming Language PASCAL. Acta Informatica 2. 335-355. (1973)

19. Jensen, K., Wirht, N.: PASCAL User Manual and Report. Lecture Notes in Computer Science, vol. 18. New York: Springer Verlag 1974.

20. Lecarme, O. and Desjardins, P.: More Comments on the Programming Language PASCAL. Acta Informatica 4. 231-243. (1975).

21. Rain, M. and others: MARY - A Portable Machine Oriented Programming Language. Mol Bulletin No. 3. 1973. október

22. Conradi, R. and Holager, P.: MARY - textbook. Runit 1979. április

23. Wirth, N.: Modula: a Language for Modular Multiprogramming. Software Practice and Experience. Vol. 7, 3-35. (1977)

24. Wirth,N.: The use of Modula. Software Practice and Experience. Vol. 7, 37-65. (1977)

25. Wirth, N.: Design and Implementation of Modula. Software Practice and Experience. Vol. 7, 67-84. (1977)

26. Wirth, N.: MODULA-2 Institut fur Informatik ETH CH-8092 Zurich, 1980 march

27. Shaw, M., Hilfinger, P. and Wulf, W.A.: TARTAN Language Design for the Ironman Requirement: Reference Manual. SIGPlan Notices 19.1978. No.9. 36-58.

28. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured programming. A.P. I.C. Studies in Data Processing. No. 8. Academic Press. London and New York 1972.

29. Cleaveland, J.C.: Meaning and Syntactic Redundancy

30. Reference Manual for the ADA Programming Language, proposed standard document, United States Department of Defense 1980. julius.

A TANULMÁNYSOROZATBAN 1980-BAN JELENTEK MEG:

101/1980 Gerencsér László - Hangos Katalin:
Diszkrét lineáris sztochasztikus rendszerek
önhangoló szabályozása

102/1980 Pásztorné Varga Katalin: Rekurziv eljárás

103/1980 Gerencsér Piroska - Szép Endre - Zilahy Ferenc
Marton Zsolt: Robotmegfogók adaptivitása I.

104/1980 Knuth Előd - Radó Péter - Tóth Árpád:
A SDLA előzetes ismertetése

105/1980 E. Knuth - P. Radó - Á. Tóth:
Preliminary description of SDLA

106/1980 Prékopa András: Sztochasztikus programozási
modellek és alkalmazásuk

107/1980 Kelle Péter: Megbizhatósági készletmodellek
és alkalmazásuk

108/1980 Almásy Gedeon: Mérlegegyenletek és mérési hibák

109/1980 Békéssy A. - Demetrovics J. - Gyepesi Gy.:
Relációs adatbázis logikai szintü vizsgálata
funkcionális függőségek szempontjából

110/1980 Gaál A. - Soltész J. - Ruda M. - Ratkó I.:
Tanulmányok a statisztikai adatfeldolgozásról

111/1980 Benedikt Szvetlána: Nem ismételhető döntéshozatal
analizise kockázattal járó esetekben

112/1980 Verebély Pál: Többprocesszoros, osztott intel-
ligenciáju grafikus rendszerek tervezési és meg-
valósitási kérdései

113/1980 V. Visegrádi Téli Iskola

114/1980 Demetrovics János: Relációs adatmodell logikai és strukturális vizsgálata

115/1980 Gergely József: Program package for sparse matrices

1981-BEN JELENTEK MEG:

116/1981 Siegler András: Egy 6 szabadságfoku antropomorf manipulátor kinematikája és számitógépes vezérlése

117/181 Knuth Előd - Radó Péter: Principles of Computer Aided System Description

118/1981 Demetrovics János - Gyepesi György: Általános függések és lekérdezéssel kapcsolatos algoritmusok relációs adatmodellekben

119/1981 Sztanó Tamás: REAL-TIME programrendszerek eseményvezérelt szervezése

120/1981 Szentgyörgyi Zsuzsa: A számitástechnika müszaki fejlődése és társadalmi hatásai

121/1981 Vicsek Tamásné (Strehó Mária): Vizsgálatok a kezdeti érték problémák numerikus megoldásával kapcsalatban

122/1981 Andó Györgyi - Lipcsey Zsolt: Sztochasztikus Ljapunov módszerek és alkalmazásaik

123/1981 Márkusz Zsuzsanna: Intelligens interaktiv rendszerek elvi problémái

124/1981 Márkusz Zsuzsanna: Logikai alapu programozási módszerek és alkalmazásaik számitógéppel segitett épitészeti tervezési feladatok megoldásához